



Tfw

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of Abhinav JALAN et al.	Confirmation No. 6956
U.S. Serial No. 10/650,041	Docket No.: 22513-03000
Filed: August 28, 2003	Examiner/Group Art Unit: Unassigned 9
For: METHOD FOR EXECUTING A SEQUENTIAL PROGRAM IN PARALLEL WITH AUTOMATIC FAULT TOLERANCE	

Hon. Commissioner of Patents and Trademarks
U.S. Patent and Trademark Office
P.O. Box 1450
Alexandria, VA 22313-1450

SUBMISSION OF PRIORITY DOCUMENT

Sir:

Submitted herewith is a certified copy of the priority document No. 884/Del/2002 on which a claim to priority was made under 35 U.S.C. §119. The Examiner is respectfully requested to acknowledge receipt of said priority document.

Respectfully submitted,

David L. Devernoe
Registration No. 50,128
Attorney for Applicant

SIDLEY AUSTIN LLP
1501 K Street, N.W.
Washington, D.C. 20005
Phone: 202-736-8818
Fax: 202-736-8711

Date: 3/30/06

THIS PAGE BLANK (USPTO)

PRIORITY DOCUMENT



INTELLECTUAL
PROPERTY INDIA

GOVERNMENT OF INDIA
MINISTRY OF COMMERCE & INDUSTRY
PATENT OFFICE, DELHI
BOUDHIK SAMPADA BHAWAN,
PLOT NO. 32, SECTOR - 14,
NEW DELHI - 110 075.

*I, the undersigned being an officer duly
authorized in accordance with the provision of the
Patent Act, 1970 hereby certify that annexed hereto is
the true copy of the Application, Complete
Specification and Drawing Sheets filed in connection
with Application for Patent No.884/Del/2002 dated
29th August 2002.*

Witness my hand this 23rd day of December 2005.

(P.K. PATNI)
Deputy Controller of Patents & Designs

BEST AVAILABLE COPY

**CERTIFIED COPY OF
PRIORITY DOCUMENT**

FORM 1
THE PATENTS ACT, 1970

(39 of 1970)

APPLICATION FOR GRANT OF A PATENT
(See Sections 5(2), 7, 54 and 135)

29 AUG 2002

1. I/we,

*Indian Institute of Information Technology, an Indian Institute, of
Nehru Science Center, Kamla Nehru Road, Allahabad - 211002,
Uttar Pradesh, India.*

2. hereby declare -

- (a) that I am/we are in possession of an invention titled "*System For Executing A Sequential Program In Parallel With Automatic Fault Tolerance.*"
- (b) that the ~~provisional~~/ complete specification relating to this invention is filed with this application
- (c) that there is no lawful ground of objection to the grant of a patent to me/us.

3. further declare that the inventor(s) for the said inventions is/are

- (a) *JALAN Abhinav, an Indian citizen, C/o Indian Institute of Information Technology, of Nehru Science Center, Kamla Nehru Road, Allahabad - 211002, Uttar Pradesh, India.*
- (b) *CHADHA Retesh, an Indian citizen, C/o Indian Institute of Information Technology, of Nehru Science Center, Kamla Nehru Road, Allahabad - 211002, Uttar Pradesh, India.*

4. I/we claim the priority from the application(s) filed in convection countries, particulars of which are as follows: NA

5. I/we state that the said invention is an improvement in or modification of the invention the particulars of which are as follows and of which I/we are the applicant/patentee: NIL

6. I/we state that the application is divided out of my/our application, the particulars of which are given below and pray that this application be deemed to have been filed on _____ under section 16 of the Act. NIL

7. That I am/we are the assignee or legal representative of the true and first inventors.

8. That my/our address for service in India is as follows:

BEST AVAILABLE COPY

& ANAND, Advocates
B-41, Jamuddin East
New Delhi - 110 013

Tel Nos.: (11) 4358078, 4355076, 4350360

Fax Nos.: (11) 4354243, 4352060

9. Following declaration was given by the inventor(s) ~~or applicant(s) in the convention country:~~

I/we the true and first inventors for this invention ~~or the applicant(s) in the convention country~~ declare that the applicant(s) herein is/are my/our assignee or legal representative

JALAN Abhinav

CHADHA Retesh

Date: 29th August 2002

10. that to the best of my/our knowledge, information and belief the fact and matters stated herein are correct and that there is no lawful ground of objection to the grant of patent to me/us on this application.
11. Followings are the attachment with the application:
- (a) ~~provisional~~/complete specification (3 copies)
 - (b) Formal Drawings (3 copies)
 - (c) Statement and Undertaking on Form 3
 - (d) Fee Rs. 5000.00 In cash/cheque/bank draft bearing no. _____, date 29/08/2002 on Citibank _____ Bank.

I/we request that a patent may be granted to me/us for the said invention

Dated this 29th day of August, 2002



Archana Shanker
of Anand & Anand Advocates
Attorney for the Applicants

To
The Controller of Patents
The Patent Office,
Delhi

Abstract

The invention describes a system and method for running a sequential program in parallel. Parts of the program to be executed in parallel to rest of the program and to each other are written in separate procedures called "parallel procedures". These parallel procedures can take arguments for input as well for output. Synchronous objects used as arguments of the parallel procedure provide automatic synchronization among all the processes. At any instance value contained in the synchronous object is same as it would have been, had all the parallel procedures been executed sequentially. If the value of the data contained in the synchronous object is not consistent, then the process accessing the object gets blocks. The system allows a synchronous object to reference other synchronous objects and maintains data consistency and linkage structure of each object. The system also provides automatic fault detection and recovery without requiring any extra input from the programmer.

THIS PAGE BLANK (USPTO)

0884-2

THE PATENTS ACT, 1970

29 AUG 2002

COMPLETE SPECIFICATION

Section 10

*"System For Executing A Sequential Program In Parallel
With Automatic Fault Tolerance"*

DUPLICATE

*Indian Institute of Information Technology, an Indian Institute, of Nehru
Science Center, Kamla Nehru Road, Allahabad – 211002, Uttar Pradesh,
India.*

The following specification particularly describes and ascertains the nature of this invention and the manner in which it is to be performed:

The patent relates to the field of parallel and distributed computing and to the field of object oriented software design. More specifically, it relates to a system for executing a sequential program in parallel with automatic fault tolerance.

5 **Background of the Invention**

Parallel computing is the use of two or more processors (computers) in combination to solve a single problem. The main challenge before a programmer writing a parallel program is to decide how to break the problem into pieces so that they can be executed in parallel and then, how to relate the fragments with each other after execution.

10

Distributed computing is a specialized form of concurrent computing in which the processing nodes (computers) are physically distributed and are interconnected. These interconnections may or may not be reliable. However the computers must cooperate in order to maintain some shared state to work on the given problem. Distributed

15 Computing harnesses the idle processing cycles of the PCs on the network and makes them available for working on computationally intensive problems that would otherwise require a supercomputer or workstation/server cluster to solve.

20 Different models can be employed to achieve Parallelism. One of them is Functional parallelism model where different tasks are executed at the same time. Another model used is Master-slave parallelism, where one process assigns subtask to other processes. A server splits large applications into small computing tasks, which are then distributed to PCs to process in parallel. Results are sent back to the server where they are collected and presented. A small software program runs on each PC, allowing the server to send and
25 receive jobs to and from that PC. In addition, SPMD (Single Program Multiple Data) where same code is replicated to each processor can also be used to attain parallelism.

The development of parallel programs is a very tedious task and involves numerous skills other than general programming skills on the part of the programmer. The development involves division of the problem into parallel executable fragments and synchronizing the
30 parallel executing processes with each other in order to produce a proper result. The programmer must also handle transfer of data from one process to another. Furthermore,

distributed systems used to run the parallel programs are unreliable and prone to system shutdowns and network failures. In order to make a system fault tolerant, a programmer has to encode the necessary complex instructions in the system to recover from a failure, which takes a lot of extra effort.

5

Parallel computing can only be used effectively if its technical issues are defined clearly and practically understood. Three basic approaches used for design and implementation of a parallel program are Control parallelism, or task parallelism, Message-passing and Data parallelism.

10

In order to add parallelism to a programming language, the language is extended, that is, the compiler is extended to recognize the new language constructs. While such new extended languages provide enhanced performance but they are limited by a lack of portability between different operating systems. Moreover, the programmer needs to learn the new language constructs in order to use them to code parallel programs. Parallel Compilers such as High Performance FORTRAN (HPF) and Data Parallel C Extensions (DPCE), are usually based on Data parallel programming model. In this model, distribution of data at a very high level is specified, and the program is then written as if the data were globally addressable by all processors and there is a single logical execution thread. This model is much easier for the programmer than attempting to directly deal with many different execution threads with data that may be on many different processors. The compiler creates and manages all of the parallel tasks and data transfer between them, based on the user's original distribution of the data.

15

20

25

30

The other approach to design and implement a parallel program, rather than using a new extended compiler, is to use Message Passing Libraries (MPL). In this model, processes communicate by sending and receiving messages. Data transfer requires cooperative operations to be performed by each process (a send operation must have a matching receive). Programming with message passing is done by linking with and making calls to libraries which manage the data exchange between processors. MPI (Message Passing Interface) is a standard message passing library providing concurrency among processes.

Parallel Virtual Machine (PVM), which provides concurrency through Message Passing libraries, is a more robust tool than MPI as it is less susceptible to failure, more scalable and allows for dynamic process scheduling unlike the fixed scheduling provided by MPI. The fixed process scheduling provided by MPI is not capable of adapting to the system changes whereas the dynamic scheduling provided by PVM can easily adapt to changes in the system configuration. The main drawback of both these libraries is that it is the programmer's responsibility to resolve data dependencies, provide fault tolerance and avoid deadlocks and race conditions using these library functions, no support is provided for automatic handling of these issues.

In Control parallelism, or Task parallelism, work is divided into multiple threads. In this model different tasks are executed at the same time. It requires all subroutines to be thread-safe. OpenMP is based on this model and uses the *fork-join* approach of parallel execution with threads. Routines for locking the data are to be used by the programmer for handling synchronization. OpenMP Fortran implementations are not required to check for dependencies, conflicts, deadlocks, race conditions or other problems that result from incorrect program execution. TOPC (Task Oriented Parallel C/C++) is a software library built on master slave model.

The same approaches of parallel programming can be used in distributed computing where the processing nodes are physically distributed and interconnected. Remote Procedure Call (RPC) is an easy and popular paradigm for implementing the client-server model of distributed computing. The semantics of RPC are almost identical to the semantics of the traditional procedure call. The major difference is that while a normal procedure call takes place between procedures of a single process in the same memory space on a single system, RPC takes place between a client process on one system and a server process on another system where both the client system and the server system are connected to a network. Like a normal procedure call, RPC is a synchronous operation, i.e., the client process is blocked until processing by the server is complete. The use of lightweight processes or threads that share the same address space allows multiple RPCs to be performed concurrently.

In RPC, arguments are passed as values but references because they cannot be passed as arguments to the remote procedure. A reference to some data is usually a pointer to that data in the memory space. Therefore, even if the pointer value is sent to the remote procedure it will be of no use as it points to the data in the memory space of the server process. Moreover usually RPCs do not allow sending of user-defined data objects to the remote procedures. This limitation of RPCs is taken care by Remote Method Invocation (RMI) a mechanism implemented in JAVA wherein the class information of an object is sent to allow the use of user-defined data objects. Moreover RMI allows the arguments to contain references as well.

10 Synchronization in RPCs is usually implemented with the help of the data locks that prevent two or more processes to access and update the data value at the same time. If a server process tries to access the data sent to a slave process, which is executing at the time of the access, then the server process gets blocked. This is done by ensuring that the slave process locks the data before accessing it. If some object is passed to a remote procedure, then the server process gets blocked when the data is accessed or is passed to some other remote procedure. This locking model ensures that no more than one remote procedure, to which the object is sent as an argument, can access it at any instance of time.

In some prior implementations, future objects are used to provide synchronization in Remote Procedure Calls. When a remote call is issued, the result of the subroutine call is stored in a 'future' object to be returned to the main program. In this way, the main program continues to run on its processor while the subroutine runs concurrently or synchronously. If the main program accesses the future object before it is available, the program simply blocks at that point to await the desired result. Future objects allow the concurrent computation of the value an object until the value is required. This allows synchronization of data among the remote procedure as well as the main program. This mechanism will not allow passing of the future value of a remote procedure as an argument to another remote procedure, thus ensuring consistency of data. During the execution of a program in parallel, fault detection and system recovery can become quite big issues in case of long running programs over processors prone to faults. Reliability of

the distributed application is encountered by PVM and other new implementations. All these systems require programmer to put in effort to put in check points in his program. The state of the program at each check point is saved so as to restore the system back to the last saved state in case of a failure.

5

US Patent 5717883 describes a method and apparatus for parallel execution of a computer program using information providing for reconstruction of a logical sequential program. However, this method has the limitation that it automatically rearranges a sequential program to allow it to be executed in parallel on a multiprocessor computer system without any provision to run the application on different nodes connected to a network.

10

US Patent 5832272 also describes a method and apparatus for parallel computation. The associates each statement of a sequential program, that would access or modify a data variable, with a serial label. This label is representative of the sequential order of execution which the program would follow when executed in parallel. However, this mode of parallel program execution is limited to multi processor computer systems with a control system in place for coordination and cannot be used to distribute applications across the nodes of the network.

15

20

US Patent 5999987 is about concurrent processing in object oriented parallel and near parallel. The patented invention provides an implementation of 'future' objects to ensure synchronization is maintained while executing a program in parallel. However, future objects have certain limitations such as they can only be used for return values of a procedure, the master process is blocked as soon as an object is to be passed as argument to a procedure and also, the value of a future object becomes available to the master process only after the remote procedure modifying it has terminated. Furthermore, future objects cannot be used for storing references to other such objects.

25

Therefore, to overcome such limitations the present invention provides an entirely new framework to implement synchronous objects that allows objects to reference other objects and maintain the corresponding linkage structure.

5 **Objects and Summary of the Invention**

To overcome the above drawbacks the invention provides a system and method for executing a sequential program in parallel with automatic fault tolerance.

10 The second object of the invention is to simplify the development of parallel and distributed programs that can be run on various platforms with the help of object oriented programming. The platform can be a multi processor computer, a high speed cluster dedicated to parallel computing, a network of computers where people do their day to day work, or even the internet.

15 Yet another object of the invention is to enable users to exploit their existing computer hardware to solve much larger problems at minimal additional cost. It is easy to observe that any computer on a network is not fully utilised at all times except during short processing bursts that require more resources for execution. The invention allows idle processing cycles of such computers to be used to execute a large application distributed
20 over the network.

These and above objectives are achieved by providing a system and method to allow a programmer to write a sequential program containing parallel procedures without any extra effort required to write a parallel program using traditional programming platforms.
25 The programmer can specify procedures to be executed in parallel using special language constructs, and the system automatically executes them in parallel with the rest of the program. The system does so by dividing the program into jobs that can be executed in parallel to each other. Data types of the arguments to the parallel procedure are changed to synchronous objects that are special wrapper classes around the regular data types
30 provided by the language platform. The system implicitly takes care of interprocess

communication and synchronization required between various executing threads and processes of the program.

5 The invention utilizes the untapped computing power of the nodes connected to the network by transferring the execution of the jobs to these nodes. These nodes act as slave nodes to the sending master node. Various threads and processes of the program can be executed on any or all the slave nodes connected to the master node. The system can automatically detect resources on the network and add them to its resource pool as and when any new resource is added to the network. The system can automatically select any
10 node on the network having sufficient free resources to execute the job or the programmer can use special language constructs to specify a particular node for job execution. The program can be assigned a low priority so that it runs as a background process not affecting other tasks being carried out on the host machine.

15 In addition, the invention provides automatic fault tolerance, i.e., no extra effort is required on the part of the programmer to write any code to recover from any fault that may occur on the network, the system does that automatically. The fault may be a processing node failure or communication link fault. If a node fails during execution of a program all the jobs given to it are transferred to some other machine unobtrusively.
20 Thus, a program can be executed on network where computers go on and off regularly.

Furthermore, a load balancing server provided by the invention ensures that the load on all the nodes executing the distributed application is equally distributed. The load balancing in the network is done automatically without any input from the programmer.

25

Brief description of the accompanying drawings

The invention will now be described with reference to the following figures:

30 **Fig 1** is the activity diagram of sending two jobs to two processing units from a single process.

Fig 2 is the activity diagram showing a slave process acting as master process sending jobs to other slave processes.

Fig 3 is the activity diagram showing the creation of a job in the master process.

Fig 4 is the activity diagram of the blocking in the master process if the unavailable synchronous object is accessed.

Fig 5 is the activity diagram of the blocking in the slave process if the unavailable synchronous object is accessed.

Fig 6 shows the concept of object ownership chain among the jobs.

Fig 7 is the activity diagram of the addition of the synchronous objects to the job.

Fig 8 is the activity diagram of the addition of the job as an owner of a synchronous object.

Fig 9 is the activity diagram of sending the job from the master process to the slave process.

Fig 10 is the activity diagram of sending the 'object info' list update from the master process.

Fig 11 is the activity diagram of sending the 'object info' list update from the slave process.

Fig 12 is the activity diagram of sending object value updates from the master process.

Fig 13 is the activity diagram of sending object value updates from the slave process.

Fig 14 shows the concept of reachability of two objects from a single object containing their references.

Fig 15 shows the concept of reachability of objects from a object through another object
5 reachable from the said object

Fig 16 shows the concept of cross-reference with an object reachable from an object by two paths.

10 **Fig 17** shows the concept of cross-reference with an object reachable from an object by two paths both through the reachable objects.

Fig 18 shows the concept of circular-reference of two objects from themselves.

15 **Fig 19** shows the concept of circular-reference of one object through two other objects.

Fig 20 is the activity diagram of receiving the 'object info' list update in the slave process.

20 **Fig 21** is the activity diagram of receiving the object value updates in the slave process.

Fig 22 is the activity diagram of updating all the jobs on receiving the ownership of any synchronous object.

25 **Fig 23** is the activity diagram of updating the job by sending the available synchronous objects to it.

Fig 24 is the activity diagram of setting the object as received after receiving the object values from its slave process.

30

Fig 25 is the activity diagram of receiving the object value updates in the master process.

Fig 26 is the activity diagram of receiving the 'object info' list update in the master process.

5 Fig 27 is the activity diagram of setting the object as sent after sending the object value to its slave process.

Fig 28 is the activity diagram showing fault tolerance with resending of the job that was assigned to a processing node that failed.

10

Fig 29 is a pictorial representation of a network of data processing systems in which the present invention may be implemented.

15 Fig 30 is a block diagram of a data processing system that may be used to execute the sequential program in accordance with a preferred embodiment of the present invention.

Detailed description of the drawings

The invention pertains to the area of parallel and distributed computing. It simplifies the way, the parallel and distributed applications are programmed. Simplification is done by obviating the need to write a parallel program in form of parallel executing processes. The system executes in parallel parts of a sequential program. Writing a parallel application involves great effort and skills on the side of programmer. Parallel programming involves extra programming load of data communication and inter-process synchronization. Program is to be divided into a set of collaborating processes or threads. Data is to be divided and then transferred from one process to another in an efficient manner, so as to maximize the performance. Synchronization between the processes is also to be taken care of. In case, the parallel execution is to be distributed over to other computers on the network, those computers must also be configured.

30 The invention provides a framework for executing a sequential program in parallel. To write a parallel application using the framework, parts to be executed in parallel are

specified in the form of separate procedures marked as 'parallel'. Therefore, the granularity, i.e. the smallest part of the program that can be divided into parallel running fragments, is the code written in parallel procedures. During the execution of the program these parallel procedures are executed in parallel to rest of the program.

5

For achieving high performance a parallel application is often distributed to various processing nodes in a cluster or on a network. For the parallel application to run successfully, these nodes need to be kept in isolation and have to be fully dedicated only for the parallel applications. A failure in any of the nodes may lead to the failure of the complete application. If the programmer has to incorporate fault tolerance he has to further program it by setting up the check points in the program and writing code for the recovery from each checkpoint on a failure between the checkpoints.

10

The invention in addition to providing a framework for executing a sequential program in parallel also provides for automatic fault tolerance. The onus of writing code for fault recovery is not on the programmer but is taken care by the framework itself. The framework automatically detects the failure and automatically recovers from the fault without any extra effort on the part of the programmer.

15

The framework automatically detects any new resource as and when added. Applications made over this framework can also be executed over a network in use for other activities. Processing nodes that are used for other activities can also be taken in. As the system is fault tolerant, these nodes can go up and down not resulting in complete failure. On each failure system performance degrades gracefully. The parts of the parallel application can be executed as low priority processes running only when a node is idle and thus not interfering with the other activities on the node. These features make the invention one of the most cost-effective supercomputer that can be put on already built network of corporate offices, research and educational institutes

25

Parallel procedure is specified at call time by specifying parallel procedure ID and its arguments to the system. ID of the parallel procedure may be in the form of address of

30

the procedure, or it can be any other ID that can be resolved to the address of the parallel procedure. It can be the name of the parallel procedure in a programming language system that allows resolving of procedures through their names. It can be any other symbol, which the programming language system is able to resolve with the procedure. It can also be any other symbol explicitly bound in the program to the procedure. A symbol can be bound with the parallel procedure using a symbol table.

Parallel procedures can be specified in the program to the framework in the following ways:

1. A job object corresponding to each call of the procedure is created. The ID of the parallel procedure and all its arguments along with their type are specified in the job object. Job object is then set to execute the parallel procedure in parallel. Sample code is as follows:

```
SynchronousInt * Argument1 = new SynchronousInt;
SynchronousFloat * Argument2 = new SynchronousFloat;
SynchronousInt * Argument3 = new SynchronousInt;
Job1 = Job.CreateJob ( ProcedureID, 3 );
Job1.AddArgument( Argument1, RETURNABLE );
Job1.AddArgument( Argument2, NONRETURNABLE );
Job1.AddArgument( Argument3, RETURNABLE );
Job1.ExecuteParallel();
```

In the code, job object is created by CreateJob() function, in which the ID of the parallel procedure is specified as ProcedureID. Then the arguments Argument1, Argument2 and Argument3 are specified by passing them to job through 'AddArgument'. The job is then set to be execute in parallel.

2. ID of the parallel procedure and all its arguments along with their type are specified to the framework through a single 'Execute parallel' function as:

/* in the main procedure */

SynchronousInt * Argument1 = new SynchronousInt;

5 SynchronousFloat * Argument2 = new SynchronousFloat;

SynchronousInt * Argument3 = new SynchronousInt;

10 ExecuteParallel(ProcedureID, 3, Argument1, RETURNABLE, Argument2, NONRETURNABLE, Argument3, RETURNABLE)

The job object is not required to be created by the programmer explicitly, the function ExecuteParallel() automatically takes care of creating it. The function is called by specifying the ID of the parallel procedure, that is, ProcedureID and its arguments Argument1, Argument2 and Argument3.

15

3. Parallel procedure may be written by making a new class derived from a common class corresponding to each parallel procedure in the program. In the class, parallel procedure may be implemented by over-ridding a procedure of the base class. Its arguments may be in the form of member variables of the object. For each call to a parallel procedure an object corresponding to the parallel procedure is instantiated which automatically creates a job for the corresponding parallel procedure. Arguments are added to the object and it is set to execute the parallel procedure. Code for the above is given as follows:

25

```
class ParallelProcedureclass1 : ParallelProcedureClass
{
    SynchronousInt * Argument1;
    SynchronousFloat * Argument2;
    SynchronousInt * Argument3;
    Void ParallelProcedure();
    ...
}
```

35

/* in the main procedure */

40 ParallelProcedureClass1 ParallelProcedureObject;

ParallelProcedureObject.Argument1 = new SynchronousInt;

ParallelProcedureObject.Argument2 = new SynchronousFloat;

```
ParallelProcedureObject.Argument3 = new SynchronousInt;
```

```
ParallelProcedureObject.ExecuteParallel();
```

5

A new class 'ParallelProcedureClass1' is derived from the pre-defined class 'ParallelProcedureClass'. The code of the parallel procedure is written in the class's overridden member function 'ParallelProcedure' and the arguments to the parallel procedure Argument1, Argument2 and Argument3 are specified as the class's member variables. The parallel procedure is executed by creating the class's object and calling the ExecuteParallel() function with the object of the class as its argument.

10

4. The task of specifying parallel procedures may also be accomplished through the compiler. Any parallel procedure is specified using a keyword like 'parallel' in its declaration and can be called in the program like any other procedure. On a call to a parallel procedure compiler automatically makes a job object corresponding to each call and executes it in parallel. Code for the above is given as follows:

15

20

```
parallel void ParallelProcedure1(SynchronousInt *, SynchronousFloat *, SynchronousInt *  
);
```

```
/* in the main procedure */
```

25

```
SynchronousInt * Argument1 = new SynchronousInt;
```

```
SynchronousFloat * Argument2 = new SynchronousFloat;
```

30

```
SynchronousInt * Argument3 = new SynchronousInt;
```

```
ParallelProcedure1(Argument1, Argument2, Argument3);
```

35

In the given code, the parallel procedure ParallelProcedure1() is specified with the keyword 'parallel' in its declaration. ParallelProcedure1() is then called like any other procedure in the program however, it gets executed in parallel.

A parallel procedure executes as a slave process to the master process from which it is called. From within parallel procedures, any number of parallel procedures can be called.

40

The calling process then acts as master to all the new processes formed. This is shown in

Figure 1, in which two parallel procedures are called resulting in their parallel execution.

During the execution of the main program (1.1) a call to a parallel procedure is made (1.2). Subsequently, a job object is created (1.3) (explained in detail in figure 3) and the job info is sent to a slave node (1.4) to be executed. The slave process that had been
5 waiting to receive a procedure from the master (1.10), receives the job info (1.11) and proceeds with the execution (1.12) till it terminates (1.13). The completion message is then sent back to the master process. Meanwhile, the master process executing sequentially comes across another parallel procedure call (1.6) and creates the job info (1.7) to be sent to slave 2 where it gets executed in the same manner as the procedure
10 execution on slave1. After completing their respective procedure executions, both the slave nodes wait (1.14, 1.19) for another job info to be sent from the master. The above diagram shows only two nodes for illustration purposes, the actual system may be comprised of at least one node each of which may be used by the master process to delegate the execution of parallel procedures. Any job can be run on any node of the
15 system on which the required resources are available at that time.

Within one parallel procedure another parallel procedure can also be called with it also executing in parallel to rest of the program. Thus a slave process of a process may be a master of others. This is shown in **Figure 2**, in which a slave process on Node1 becomes
20 a master of another slave process on Node2 as it transfers the execution of a procedure to the Node2 slave. The master node sends job info (2.4) about a parallel procedure to be executed to slave Node1. Here the slave process that had been waiting for the info (2.6), begins its execution (2.8) after receiving the info (2.7). During the procedure execution, a call to another parallel procedure is encountered (2.9). Therefore the slave process,
25 creates a new job info (2.10) for this procedure and sends it to slave node 2 (2.11). Slave process on Node2 receives the info (2.16), executes the procedure (2.17) till it terminates (2.18). Subsequently, the slave process on Node1 completes the execution of the procedure sent to it by the master (2.14), subject to the condition that it does not have to halt execution due to getting blocked.

30

On a call to a parallel procedure a job object is created and synchronous objects passed as arguments to the parallel procedure are added to the job as shown in detail in **Figure 3**. In the figure, once the framework is signaled to execute a new job, the new job object created (3.1) and the ID of the parallel procedure is passed (3.2) to the job object. Then
5 the arguments of the parallel procedure are specified to the job object (3.3) and these arguments objects are added to the job object (3.4). The objects that are available to the master are sent along the ID of the parallel procedure to the slave process waiting for the arguments. The job object is updated in the slave as new objects become available (3.5).

10 Data transfer between the master and slave processes is done through arguments passed to the parallel procedure. Data needed in the parallel procedure is put in the argument variables in the master process. Any data needed from the parallel procedure is written back on the argument variables in the parallel procedure. Changes made in the argument variables are reflected back in the master process. Arguments can be only for input [in] as
15 well as for both input and output [inout]. The [in] arguments are non-returnable while the [inout] arguments are returnable. [inout] arguments are updated back on the master process while [in] arguments are not.

For the program to give correct results, values of variables at any instance of accessing
20 them, must be consistent with values they would have, had the program been executed sequentially with no parallel procedure executing in parallel. In this model, synchronization of data between master and slave processes is also to be taken care of. Variables sent as arguments in the parallel procedure may not have right values in them at all instances during the execution. For example, a parallel procedure is called and its
25 returnable argument is modified in it. As the parallel procedures execute in parallel to rest of the program, the argument variable can be accessed outside the parallel procedure before its completion. It can be accessed in the master process or it can be sent to some other parallel procedure also. Now the value of the variable is inconsistent with the value, it should have been, had the program been executed sequentially. Its value must be first
30 updated back from the modifying parallel procedure, and then it must be accessed in other threads. If through some mechanism the thread is blocked during accessing the

value till its correct value is put in place, and correct value is put in the variable as it gets evaluated in other thread, then data consistency is maintained

Following are the cases to be handled to achieve synchronization in the above model of parallel execution of a sequential program:

1. Master process accesses the data value that is sent as returnable to the slave process still in execution. This is the case when master process sends some returnable data to the slave process and then tries to access it even though it does not have the correct value as the parallel procedure is still in execution and can change the returnable data.
2. Slave process accesses data value when it has not received the correct value from the master process as master was not having it at the point of calling the parallel procedure. Suppose returnable variable A is passed to job J1 and then to job J2. Now, since J1 and J2 execute in parallel, any changes made to A do not get reflected on J2 and therefore, J2 does not has the right value of A.

The framework automatically accomplishes synchronization and maintains data consistency without requiring any extra input in the program. No effort regarding data dependencies or synchronization between different processes is required to be made by the programmer, only the way the arguments are passed to the parallel procedure is changed. Arguments are passed in the form of special synchronous objects encapsulating the data carried by the arguments. These synchronous objects act as wrapper objects around argument data.

Synchronous objects are instances of 'SynchronousObject' class. If data of type 'signed integer' is to be sent as an argument to a parallel procedure, an object of SynchronousInt class is instantiated and its reference is passed as argument to the parallel procedure. SynchronousInt is wrapper class of 'signed integer'. New SynchronousObjects classes are made by deriving new classes from the SynchronousObjectBase class. The framework provides with SynchronousObjectBase class, inbuilt wrapper classes of some basic data types derived from SynchronousObjectBase class. Custom SynchronousObjects classes can be made by deriving them from other synchronous

object classes. For example, if a complex number is to be sent to the parallel procedure, a new class part is made by deriving it from the SynchronousObjectBase class containing a real and an imaginary part.

5 Direct access of data contained in a synchronous object is not allowed. Data contained in synchronous object is accessed or modified only through object procedures. Every object has an access lock associated with it that can be any synchronization object like a semaphore or any other synchronization object provided by the operating system. Every time a process accesses the object, the access lock is locked and unlocked by the process
10 depending on the availability of the object to the process. If the object is available, process locks it to prevent the object's parallel use by any other process unlocking the object after use. If object is not available, process gets blocked when trying to lock the access lock and is resumed back when the object becomes available. Thus on accessing an argument in master or slave process, it is ensured that accessing process gets the right
15 value. The right value is one that would have resulted if the program had been executed sequentially.

Figure 4 shows the blocking and resuming of the thread executing the master process when accessing data sent as argument to parallel procedure to a slave node. When a call
20 is made to a parallel procedure (4.1) in the master process, the required job info is sent to the slave node which starts a slave process execution (4.6) to execute the procedure. If during the execution of this procedure on the slave an object passed as an argument to the slave is accessed by the master (4.2), then the thread executing the master is blocked (4.3). The thread resumes (4.5) only after the slave process terminates (4.7) and data
25 values are updated in the master (4.4). **Figure 5** similar to Figure 4, shows the blocking and resuming of the thread executing the slave process 1 when accessing data sent as argument to parallel procedure to a slave process 2. In Figure 5, the slave process gets blocked (5.5) after trying to access an object (5.4) whose value has not been received by the job. It is able to resume execution (5.8) only after the master process updates the
30 object's value with the correct value.

In every job, a list of a 'Object-info' structure is maintained. The structure contains a reference to the object. It also contains the status of the object with respect to the job, that is, whether object values have been transferred to the slave process or not. If object is passed as returnable, then structure also stores if object's ownership have been 'received' back from the slave process or not. Separate 'Object-info' lists are maintained for returnable and non-returnable objects. A single list can also be used with additional information about each object's type (returnable or non-returnable) in each element of the list. Each Object-info also has a pointer to the next owner of the object in ownership queue of the object. The pointer is to the object's Object-info in the next job. If there is no next owner then pointer is null. Object itself contains the pointer to the Object-info of the current owner, which is the first element in the ownership queue. Ownership queue of an object is nothing but this linking of the object info corresponding to the first owner with the object info corresponding to second owner and so on. In the queue jobs get inserted and deleted by just changing the sequence of linkage of object info.

Figure 6 elaborates the concept of "Object - info" lists. In the figure, each job has two "object-info" lists associated with it; one list is used for objects passed as returnable and the other for the non-returnable object type. The sample synchronous object (6.1) is passed as returnable to job1 (6.2) and thus, the returnable "object-info" list maintains a pointer to it. Also associated with the object is the Object Buffer (6.3) which holds the actual value of the object. The Object Buffer is used for providing fault tolerance, this use is discussed later in the specification. The same synchronous object (6.1) is passed as both returnable and non-returnable to job2 (6.4) and therefore, it has references in both returnable and non- returnable "Object - info" lists of the job. References in both the list maintain a pointer to the same Object Buffer (6.5) which stores the actual value of the object. Furthermore, the object (6.1) is passed as only non-returnable to job3 (6.6) and job4 (6.8), hence it is referenced only in the non- returnable list of these jobs. The "object - info" lists of both the jobs refer to the common Object Buffer (6.7) because the object is passed as non- returnable and only a read only copy of it is required by the jobs.

Objects that are arguments of the parallel procedure must be added to the job object before it is sent to the slave node for execution as shown in **Figure 7**. The job determines whether any previous occurrence of the object (7.1) is found in the job info list. If an occurrence is not found (7.2), then the object's info is added to the job's "object-info" list. But if an occurrence is found (7.3), then the object info of the previous instance is used. In both cases, the next step is to check the object's association (7.4), i.e., determine from which intermediate object the object has been reached. Subsequently, the object's association information is added to the object info list (7.5) and the job is added as an owner in the object (7.6). If the object is Null, then the job is not added as an owner.

In the slave processes, object is initially 'source owned', that is, it is not available to the slave process. Only after the slave process receives object values from the master process (source) it becomes available to the process for accession and modification. In the slave process, the object can be passed to a new job before object gets value from the master. After object values are received from the master process, object's ownership is transferred to the first job in the queue without object becoming available to the slave process. The object becomes available to the process only after queue gets emptied. The process gets blocked, if it accesses the object before object becomes available to it. At any instance of execution a synchronous object is available to a single process only.

After an object has been added to a job, the job takes the ownership of the job after determining the current ownership status of the object as depicted in **Figure 8**. The first step is to check the object state, that is, whether the job is available or not (8.1). If the object is available, the object is added to job's "new object available" list (8.2), followed by setting the object as "job owned" (8.3) and locking the accession lock of the object (8.8). If the object is not available, then it is determined whether the object is owned or not (8.4). If the object is not owned, then the object is removed from the "objects to be freed list" (8.5) and the object is added to the "new objects available list" (explained later in the specification) of the new job (8.12). If the object is owned, then a further determination is done about whether the job is "source owned" or "job owned". If the object is "source owned", then insert the job in the ownership chain of the object (8.10). However, if the

object is "job owned", then also the job is inserted in the ownership chain of the object (8.7). The position of insertion is obtained (8.9). If the job is inserted in the first position, then the object is removed from the "new objects available" list in the second job (8.11) and put in the corresponding list in the new job (8.12).

5

After getting the ownership, object values are transferred to the slave process and the child object in the slave process is updated with the right values and becomes available to the slave process. Objects may be passed as returnable or non-returnable arguments to the slave process.

- 10
- If an object is passed as returnable, then it becomes unavailable to the master process and becomes available to the slave process. After the completion of parallel procedure the value at master process is updated and object becomes available to the master process again., i.e., the job releases the ownership of the object on the completion of the parallel procedure after parent object in the master process has been

15

updated with the value from the child object in the slave process

- If an object is passed as non-returnable then it remains available to the master process and it is not updated back on the master process after completion of the parallel procedure, i.e., the job does not take the ownership of the object and ownership is released same moment.

20

If an object, already owned by a job is added to a new job, the new job is inserted in the queue of jobs to get the ownership of the object. The job is inserted irrespective of whether object is added as returnable or non-returnable. After an object's ownership is released by a job, its ownership gets transferred to next job in the queue. As ownership is

25

passed on to the next job, object value is transferred to corresponding slave process. After parent object in master process gets back the object's ownership from first slave process, object's ownership is transferred to second slave process. In the process of transferring of the ownership, object is not made available to the master process until the ownership queue of the object does not become empty. In the queue, if the object is added as

30

returnable in the next job, next job releases the ownership after completion of its slave

process. If the object is added as non-returnable, the job does not keep the ownership and transfers it to the next job in the queue.

After the creation of job object, it is sent to the slave node that is selected to execute the process. **Figure 9** depicts an activity diagram of sending a job from the master process to a slave process. Slave processes is created on a slave node and a job header (9.1), containing JobID, parallel procedure ID or address, argument count etc., is passed to it by the master process. After the job header, the information of the object (9.2) sent as argument to the process is sent, followed by this the object's argument information (9.3).

This information notifies the slave process about which objects in the job are arguments. Following this the values of the objects available to the master process are sent to the slave process (9.4). The ownership of the object is changed depending on whether the object is returnable or non-returnable.

While transferring the job from master to a slave node, "object – info" list update is also sent across as shown in **Figure 10**. The master sends the object information count (10.1) to the slave informing it about the number of objects to be sent as arguments of the job. Then the master checks if the object information (10.2) for all the objects has been sent or not. If any object remains to be sent, then select the next object to be sent (10.3) and it is

determined whether the object has been received from the master process (10.4) or not. If the object has been received then, the object's class info is sent (10.4) to the slave. If the object is Null, then the class info is not sent. If the object has not been received from the master, then only the object's position in the job is sent (10.6). Class-info is a structure associated with every SynchronousObject class and contains the method to create an instance of the class at fly. In the definition of every SynchronousObject class a procedure is added that can be called to create an object using class' default constructor. This procedure and Class-info object can also be added in the class definition by adding a single macro. Sample code for a macro implementation can be as follows:

```
#define DYNAMIC_CREATE(className)\
    static ClassInfo ClassInfo(className::CreateObject);\
```

```

virtual ClassInfo& GetClassInfo()\
{\
    return ClassInfo;\
}\
5 static ClassName * CreateObject()\
{\
    return new ClassName();\
}

```

- 10 The Class-info of a class contains the address of this procedure that can be used to create the object during program execution. A flag indicating whether the object is Null or not is also sent to the slave process. Corresponding **Figure 11**, shows the transfer of “object – info” list update from one slave process to another slave. This process is very similar to the one depicted in Figure 10, with the additional step (11.5), wherein the object’s
- 15 position in the job is also sent to the slave process. In both these cases, if the object is NULL then the class info is not sent to the slave.

- After each synchronous object’s class info has been transferred to the slave process and child synchronous objects created, data of the objects available to the job is sent. Object
- 20 values of the rest of the objects are transferred to the slave process as and when the master process gets their ownership. The process of transferring object value updates is shown in **Figure 12**. The master process checks whether the values of all the available objects have been sent (12.2) using the object value update count (12.1), which holds a count of all the object to be transferred. If some object values are still to be sent, then an
- 25 object info is taken from the “objects to be transferred “list (12.4) and the object’s position and type (returnable or non-returnable) (12.5) are sent to the slave process. The “objects to be transferred “ list is maintained in every job and holds the references to any newly available objects whose value is to be sent to the slave process. The contents of the object are serialized (12.6) and its value and link info is sent (12.7) to the slave process. If
- 30 no more updates are to be sent to a slave process, then the “objects to be transferred” list is emptied (12.3).

Corresponding **Figure 13**, shows the process of transferring object value updates from one slave process to another after the ownership of the object has been passed to it. After sending the object value count (13.1), it is determined whether the values of all the available object in the "objects to bet transferred" list (13.2) have been sent or not. If all the values have been sent, then empty the "objects to bet transferred" list (13.7). If the values remain to be sent, then an object info is taken from the list one at a time (13.3) and its position is sent (13.4). The contents of the object are serialized (13.5), following which its value and link info is sent (13.6). The object value can be extracted during the extraction of the linked objects and storing the extracted value in the buffer resulting in a single call to Serialize procedure.

In order to transfer object values from one process to another, every synchronous object needs to have procedures for serialization and deserialization. Serialization is the process of copying objects on a data stream. Deserialization is the process of updating the object by copying data from a data stream to the object variables. In the 'serialize' procedure data in object's variables is copied to the argument stream one by one. In the 'deserialize' procedure object is updated back by copying data from the argument stream to the object variables in the same order. Both serialize and deserialize take an object of argument stream as arguments. Argument stream acts as an abstraction to the data stream. Serialize is called when sending the value of a synchronous object and deserialize is called when receiving and updating the value of the synchronous object.

After transferring the value of the object its status is set as 'sent' in the master process. **Figure 27** shows the activity diagram of setting the object as sent after sending the object value to its slave process. First, a check is made to determine whether the object has a next owner (27.1) or not. If the object has a next owner, its ownership is transferred to the next owner (27.4) and it is added to the next owner's "new object available list" (27.5). The status of the objects becomes 'sent' in the present job. If the object has no next owner, then object's current owner is set as Null and the object is added to "Objects to be freed list" (27.5).

Argument objects' references are put on the process stack and parallel procedure is called after the object's class info(s) has been transferred to the slave process. After the completion of the parallel procedure, value of data in returnable object is sent back to the master process to update back the parent object using the same method of object serialization and deserialization.

In the slave process, a 'child synchronous object' of each synchronous object passed as arguments is created. Child object in the slave process is a copy of the parent object passed as argument in the master process. The slave process is able to access the child object only when its values are updated from the master process. Parent object is serialized in the master process and updated in the child process in the slave process. All operations on the argument objects are done on slave process's local copy. If the object is passed as returnable, then on completion of the parallel procedure, object in the master process is updated back. Child object is serialized in slave process and parent object in master process is updated back.

A returnable object may be transferred back to the master process before the completion of the parallel procedure. If through some means, the last instruction where object gets modified is detected, the object can be transferred back right after that instruction is executed. This position can be determined by allowing the programmer to specify it. This can be done by calling a procedure of the object or can be determined by the compiler itself. If not specified, the end of all the operations on the object is determined with the end of the parallel procedure.

This property of the synchronous objects does not allow any data inconsistency in the program. As a result the parallel procedure gets executed in parallel, to the extent it can, till a data dependency with other parallel executing procedures is encountered. For best performance data dependence must be minimum and if possible as late in a parallel procedure as possible so that code before the dependence gets executed in parallel.

In many cases, a process may get terminated before it transfers object values to other processes (slave or master). Following are the possibilities:

1. Slave process terminates after sending object received from its master process to its slave process for modification. The object must now needs to be updated back on the slave process and then to its master process. Suppose a returnable object A is sent to job J1 from where it is sent further to job J11 and the process executing J1 terminates before J11 completes, then the modification in the value of A in J11 do not get reflected back in the master process of J1.
2. Master process terminates or object goes out of scope before object value to be transferred from its one slave process to another. This case is only possible when the master process does not access the object after call to the parallel procedure, else the master process would have got blocked till the object value is received from the slave process. It may lead to blocking of slave processes indefinitely. Suppose a returnable object A is sent to job J1 and then to job J2, but the value of A is not yet transferred to J2. Now, if the master process is terminated before transferring the updated value of A by J1 to J2, the value of object A gets never updated in J2.

Slave process terminates before receiving an object value from its master process. This case is only possible when the object is not accessed in the parallel procedure, else the slave process would have got blocked till object value is received from the master process. Suppose a returnable object A is sent to job J1 but its value is not transferred yet. Now, if A is not accessed in job J1 and it terminates before receiving the correct value of A from the master process, no process remains to receive value of A and then to transfer it back to master process (so that job releases its ownership).

For the aforementioned reasons, a synchronous object is not deleted until all object transfers have taken place through the process and memory management of the synchronous objects is not handled by the programmer but by the system automatically. Synchronous object are not created on stack as they are required to exist even after termination of a process and after the end of program block scope in which synchronous object is created. One way to accomplish the required memory management scheme is to

allow the programmer to explicitly mark the synchronous object as 'over' when it is no longer needed. The system automatically deletes the object after all the object transfers have taken place and object has been marked as "over". Some programming language systems provide such automatic garbage collection, wherein an object is garbage collected when no longer in use. In language systems, where automatic garbage collection is not supported, garbage collection can be done using what are called as "smart pointers". Each object maintains a reference count of the object. Reference count is number of references (smart pointer) to the object present at any instance of the execution. After all smart pointers to the object get deleted, object also gets deleted. Initially when the object is created its reference count is initialized to one. Whenever there is a new reference to the object the reference count of the object is incremented by one and whenever the reference to the object gets deleted the reference count of the object is decremented by one. The object gets deleted when the reference count of the object becomes zero. References to the object, whose value transfers have to take place, are maintained in the job object and therefore object does not get deleted till all object transfers have completed. The operations that can be done on a reference (smart pointer) of the synchronous objects in the slave process include:

1. Nullifying the reference. The reference count of the synchronous object is decremented. Reference count is also decremented when a smart pointer gets destroyed. A smart pointer gets destroyed in many ways. The pointer may go out of scope. The pointer may be explicitly deleted by the programmer or it may get deleted when data structure containing it gets deleted.
2. Changing the reference to some other already existing synchronous object. In this case, the reference count of the object to which the pointer was referring to is decremented while the reference count of the object, the said object is now referring to, is incremented.
3. Changing a null reference to a new synchronous object. In this case, the reference count of the new synchronous object is incremented.

However, the above model is not capable of handling sequential programs wherein, arguments may contain references or pointers to some other objects. If an object containing references to the data outside the object is sent to another process, then the data must also be transferred along, as the data may be accessed in the slave process through the object. This transfer of data poses the problem of synchronization of this data also. This data can be referred through the references that are not in the object. After the data has been transferred to the slave process, one of its references remains in the first process. Two copies of the data, one in the master process and other in the slave process, can be modified independent of each other resulting in inconsistencies in data in one of the processes involved. However, if this data is also in the form of synchronous object, the synchronization problem gets solved automatically.

Synchronous objects may therefore contain references to other synchronous objects only. An object is reachable from another object, if the reference to the former object is contained in the latter or any other object reachable from the later object. In simple terms a synchronous object A is reachable from another synchronous object B, if object A can be accessed in the program through object B. For instance, in **Figure 14**, objects B (14.2) and C (14.3) are reachable from A (14.1). In **Figure 15**, references of G (15.4) and H (15.5) are contained in E (15.2) and F (15.3) respectively and thus are reachable from E (15.2) and F (15.3). E (15.2) and F (15.3) in turn are reachable from D (15.1) and so G (15.4) and H (15.5) are also reachable from D (15.1) also.

When an object is added to a job all the objects reachable from it are also added into the job. Similarly, if an object is transferred from one job to another, all the objects reachable from it are also transferred along with it. In addition, as and when a synchronous object becomes available to the job, it is transferred to the slave process. An object, reachable from a returnable object is treated as returnable, and an object, reachable from a non-returnable object is treated as a non-returnable in the job. Object values of all the reachable objects are sent back to the master process on completion of the parallel procedure, and corresponding objects in the master process are updated and their

ownership is released by the job. Ownership of non-returnable objects is released after sending their values to the slave process.

When adding the object to the job, the job object does not have prior information about the references contained in the object, therefore it uses serialize and deserialize procedures of the object to gain this information. In serialize and deserialize procedure of a synchronous object, the references (smart pointers) to synchronous objects contained in it are also passed to the argument stream in the same order as the objects themselves. When the objects' serialize procedure is called, references to synchronous object(s), having their references in the object, are added to the argument stream. Argument stream does not copy the reference value (possibly the address of the object in the memory) to any data stream, rather it passes the reference to the job. The object is added to the job in form of a new entry in the object info list and its serialize procedure is called again to gain information about the references further contained in this object. Thus all objects that are reachable from the argument object get serialized recursively. After all objects reachable from one argument object have been added, the same process is repeated for remaining objects. The following sample code illustrates a new synchronous object class that contains references to other synchronous objects.

```
20  Class NewSynchronousObject : BaseSynchronousObject
    {
        int Var1;
        float Var2;
        int Var3;

25  void serialize(Stream ArgumentStream)
    {
        ArgumentStream.serialize ( Var1 );
        ArgumentStream.serialize ( Var2 );
30  ArgumentStream.serialize ( Var3 );

    }

    void deserialize(Stream ArgumentStream)
```

```

{
    ArgumentStream.deserialize ( Var1 );
    ArgumentStream.deserialize ( Var2 );
    ArgumentStream.deserialize ( Var3 );
}
...
}

```

10 In the code given above the class `NewSynchronousObjectClass` is derived from `BaseSynchronousObjectClass` and contains other synchronous objects `Argument1`, `Argument2` and `Argument3`. In `serialize()` function, all the synchronous objects are copied in the argument Stream and in `deserialize()` function, all the objects are updated back by copying them from the argument Stream.

15

Figure 20 shows the activity diagram of receiving the “object info” update in the slave process. First, the object info count is received (20.1) informing the slave about the number of objects passed by the master as arguments of the procedure. Then it is determined whether the class info for all the objects has been received or not (20.2). If all the info has not been received, then next step is to check whether the object received is New (20.3). If the object is not New, then the object’s position is received in the slave process (20.10) and the object is added to the “returned object” list in the job (20.9). If the object is new, then it is determined whether the object is Null or not (20.4). If the object received is not Null, then receive object’s class info sent by the master (20.5). Class info of each object in the list is transferred to the slave process in the same order, as in object info list, along with parallel procedure ID. After receiving the list, objects are created from the class info list (20.6). The new object’s status is changed to “source owned” (20.7) and the object’s accession lock is locked (20.8). A reference to the object is stored in same order in a ‘received objects list’ (20.9). The argument stream updates the object references to the newly created objects using the deserialize procedure. If the object received is Null, then the object is added to the ‘received objects list’ (20.9).

After sending the object info updates as shown in Figure 11 and 13, the master process sends the object value updates to the slave. Value of all the objects available to the master process at the point of calling the parallel procedure is sent right away while, remaining object values are transferred from master to the slave as and when they become available to the master process. The slave process receives these value updates as shown in Figure 5 21. First, the object value update count is received (21.1), informing the slave about the number of objects to be updated. Then a check is made to determine whether all the updates have been received or not (21.2). For the objects whose updates still remain to be received, the object's position in the job is received (21.3) and its position is added to the "received object" list. Next, the object's value (21.4) is received along with its link info. 10 This link info is the list of positions in the object info list of all the objects that contain references to the present object. Object value is then deserialized (21.5) to the corresponding object using the object's position sent by the master. Finally, the object is set as received (21.6) and ownership status of the object is changed to job owned rather 15 than source owned.

After the completion of parallel procedure, each object value along with the object position in the 'return object list' is transferred back to the master process as object becomes available. After the completion of parallel procedure or the end of all the 20 operations on the object, its object value along with the object position in the job is transferred back to the master process as it becomes available.

A reference to a synchronous object is link between the two objects. Link is directional and directed towards the referred object. When an object gets transferred from one 25 process to another along with objects reachable from it, the linkage structure of the synchronous object is maintained, i.e., all the paths through which one object was reachable from another is maintained. To maintain this linkage structure, cross-references and circular references in the objects have to be taken care of. If the objects are assumed to be nodes of a graph and links connecting these nodes to be directional edges, then the 30 cross reference corresponds to two paths to reach from one to another. Circular reference corresponds to a loop in the graph. A special consideration while handling cross-

references in an object, is to avoid sending an object reachable from other objects through more than one path, more than once to the next job. Cross-reference can be illustrated in Figure 16 and Figure 17. In **Figure 16**, object C (16.3) is reachable from object A (16.1) by two paths one through B (16.2) which is reachable from A (16.1) and the other directly reachable from A (16.1). In **Figure 17**, object F (17.3) is reachable from D (17.1) through two paths, one through E (17.2) and the other through G (17.4), both E (17.2) and G (17.4) being reachable from D (17.1) directly as they are contained in it.

Further complications in the circular reference linkage structure of an object are illustrated in Figure 18 and Figure 19. In **Figure 18** both objects A (18.1) and B (18.2) become reachable from each other. A (18.1) is reachable from B (18.2) directly as its reference is contained in B (18.2) and B (18.2) is reachable from A (18.1) directly as reference of B (18.2) is contained in A (18.1). In **Figure 19**, object C (19.1) becomes reachable from itself due to the circular reference. The object D (19.2) is reachable from C (19.1) as its reference is contained in C (19.1). Furthermore, E (19.3) is reachable from D (19.2) and in addition, C (19.1) is again reachable from E (19.3) thereby completing the circular reference.

Adding objects to a job with cross-references or circular references is similar to adding an object as argument to the job more than once. If an object is added twice to a job with same type (either both as returnable or both as non-returnable), only one copy of object must be formed in the slave process. All the references of the object (in the slave process) must point to the same copy of the object. In case of cross-reference and circular reference, the object is added to job twice. In order to incorporate it, object info also stores the first occurrence (of same type) of the object in the job (object info list). When transferring object info list, first occurrence of the object is also transferred along with their class info. If the first occurrence position is not the current position of the object, new instance of the object is not made in the slave process, rather in the 'received object list' reference in the first occurrence is copied in the current position. During deserialization process, same object is linked and thus linkage structure is maintained. If object is added twice with different types then two different copies are made in the slave

process, as the two types have very different functionality and cannot be mixed by the framework.

Another problem that needs to be addressed is when the programmer changes the linkage structure of the objects. The above model of recursively adding the jobs fails, when the references contained in the synchronous objects are changed. The model adds all the objects reachable from it, irrespective of whether object is available to the job at that point or not. If in a slave process, references in synchronous objects are changed, then these changes must also get reflected in the master process and all the slave processes called after it. However the above model fails at two points. First, it does not reflect linkage changes in the slave process to the master process. No information regarding any change of linkage structure is passed back to the master process. Values of objects received from the master process are sent back only with the object's position in the job. A new object can be created, and a reference in a synchronous object that is received from the master process, can be changed to the new object replacing the previous object. In this case, neither the new object's class info, nor the object's value is sent back. Second point of failure in the above model is that, even if linkage information is sent back and corresponding new objects are created and linked in the master process, changes in the linkage structure may still not get reflected in the job created after the one in which changes have been made. For instance, a synchronous object A containing a reference to another object B is passed to two jobs J1 and J2 as returnable. A is now unavailable to J2 till J1 releases its ownership. In J1, reference to B may be modified to a new object C created in J1 only. However in J2, A gets added and along with it B also gets added resulting in inconsistency in the linkage structure of A. Modification in the references may include:

1. A reference to a synchronous object contained in an object may be made null.
2. A reference to a synchronous object contained in an object may be set to an object received from master process.
3. A reference to a synchronous object contained in an object may be set to an object created in slave process itself.

If a synchronous object is not available, then the process having object's ownership may modify references to synchronous objects contained in it. Therefore, if a synchronous object is not available to the job, objects having their references in it are not added to the job till the object becomes available to the job.

Job object maintains a 'New objects available list'. An object already added to the job gets added to the list as it becomes available to the job. When a new object is added to the job, the object also gets added to the list if the object is available to the job. Else, when object becomes available to the job, it gets added in the 'new objects available list' of the job. After argument objects are added to the job, the system checks for objects in 'new objects available list'. For each object in the 'New objects available list', it adds all the objects to the job that have their references in the object (that is present in the 'new objects available list') and have yet not been added to the job (to check for circular and cross references). Before the addition of objects (to the job) having their reference in an object present in the 'new objects available list', the object is deleted from the list. This process of adding and removing objects continues till the 'new object available list' becomes empty.

Master process maintains a list of jobs that are not complete at any particular instance. It also maintains a 'jobs to be updated list'. 'Jobs to be updated list' contains jobs that have at least one object in their 'New objects available list'. The system updates all the jobs one by one removing their entry from the list as they are processed. Job updating comprises the process of emptying its 'new object available list' by adding the objects to the job. Jobs are numbered in the order in which their parallel procedures are called. Numbering is done during the creation of the job object. The updating of the jobs is done in the order in which their parallel procedures are called, that is, in the order in which they are numbered. Any time an job's 'New objects available list' is empty and an object gets added to it, then the job also gets added to the 'jobs to be updated list'. Therefore, the job can get added to the list only when it is created and when new objects that are available to the job are added to it, or when an object's ownership is received by a

process (master or slave). After a process receives an object, ownership of the synchronous object is transferred to the next job in the ownership queue and objects get added to the 'new object available list' of the new owner job. With this, the new owner job also gets added to 'jobs to be updated list'. To explain the process an example is provided as follows: Suppose a synchronous object A containing reference to another synchronous object B is returned to the master process. If object B is already added to job J2 and object A to job J3, with J2 being called before J3, then B gets added to the 'New objects available list' of J2 and A gets added to the 'New objects available list' of J3 and both J2 and J3 are added to the 'Jobs to be updated list'. While updating jobs, J2 gets updated before J3 (as J2 is called before J3). When J3 is updated, object B is added to the job J3 and remains unavailable to J3 till J2 releases its ownership. However, if the link between A and B would have broken in the first job, then object B would not have been added to the job J3.

Similarly, suppose a synchronous object A containing a reference to another synchronous object B is returned to the master process. If object A has already been added to job J2 and object B to job J3, with J2 being called before J3, then A gets added to the 'New objects available list' of J2 and B gets added to the 'New objects available list' of J3 while both J2 and J3 get added to the 'Jobs to be updated list'. While updating jobs, J2 gets updated before J3 as it is called before J3. When J2 gets updated, object B also gets added to J2 as it is referred in object A which is in J2's 'new object available list'. When the object B is added to job J2 it becomes unavailable for job J3 and is therefore removed from 'new object available list' of J3. Thus object B's value is not transferred till its ownership is released by J2. This above example demonstrates that a second job called after the first one, cannot have the ownership of an object reachable from another object still owned by the first job. Thus, a job must not release ownership of an object till it releases the ownership of all the objects from which the object is reachable. For example, synchronous objects A and B are sent to job J1 with B being reachable from A. So the job does not release the ownership of B till it releases the ownership of A. If ownership of B is released before A, then modification to B is possible in J1 through A as B is reachable

from A. But no such modification gets reflected back in the master process resulting in inconsistency of data in J1.

Master process also maintains an 'Objects to be freed list'. When a returnable object is received back in the master process, it is not freed, i.e., object does not become accessible to the process even if no next owner exists. Instead, it is added to the 'Objects to be freed list'. After receiving the object, all the jobs in 'jobs to be updated list' are updated. When no job is left to be updated, all the objects in 'objects to be freed list' are made available to the master process. In the process of updating jobs, the object may be added to another job. There is also a possibility along with this returned object some other synchronous objects containing reference to this synchronous object may also be returned. If any of them has a job as its next owner, then this object will also get added to the job and become unavailable to the master process. If an object is present in 'object to be freed list' and gets added to some other job, then it is removed from the 'objects to be freed list'. After all the jobs are updated and object remaining in the 'objects to be freed list' are made available to the master process. This process is illustrated in the activity diagram in **Figure 22**. To begin with, a job is taken from the "jobs to be updated list" (22.1) and its reference is deleted from the list (22.2). The job is then updated (22.3) and the status of all the corresponding objects in the "objects to be freed list" is set to "not owned". All the objects in "objects to be freed" list are set as "available" and their accession locks are unlocked (22.6). All references of objects in the "objects to be freed" list are removed emptying the list (22.7). The above process is repeated for every job in the "jobs to be updated" list till the list is not empty.

To further elaborate the process an example is provided as follows: Suppose synchronous objects A and B are sent to job J1 as returnable, and object A is also added to job J2. In job J1, B becomes reachable from A, that is, its reference is put in A. After job J1 releases the ownership of A and B, B is added to the 'objects to be freed list' and is not made available to the master process. However, object A is added to the 'New objects available list' of J2. So J2 gets added to the 'Jobs to be updated list'. When J2 is updated,

object B also gets added to job J2 along with A resulting in it being removed from the 'objects to be freed list'. Therefore, B is not made available to the master process.

When an object is received back in the master process, all the jobs in 'Jobs to be updated list' are updated in the order in which their parallel procedures are called. If a synchronous object is not sent to a slave process, it can't be referred in any of the objects owned by its job, else it would be owned by the job. This ensures that if a synchronous object gets added to a job, then it could not be modified in any other parallel procedure called before the calling parallel procedure of the job, thereby maintaining the consistency of data in synchronous objects. It also allows for modifications to be done in the references to synchronous objects contained in a synchronous object.

The objects referred from an unavailable object are not added to the job till the object becomes available. Therefore, objects cannot be added to the job recursively (depth first). First, all the objects referred from an object in 'new object available list' are added to the job. Serialize procedure is used to add the references to the synchronous objects contained in the object to the argument stream. However, the serialize procedure of the newly added object is not called until they become available to the job. When the new object is available to the job, it is added to 'new object available list' and the framework now calls the serialize procedure of the object in 'new object available list'. (This is effectively adding of the objects in breadth-first manner.)

Figure 23 shows the activity diagram of updating the job by sending the available synchronous objects to it. The job checks whether any object is available in the "new object available list" (23.1). If there is an existing object, it is taken from the list (23.2) and its reference is deleted from the list (23.3). The references of all the objects referred to in this object are also added to the argument stream using serialize procedure (23.4) and the object is then added to the "object to send list". The status of the object is changed to "sent" in the job.

Order in which objects are added to the job is not fixed, as it depends on whether an object is available or not at the point of its addition to the job. It is also subject to changes in the linkage structure of the object. Since order of addition of objects is not fixed, an extra data structure containing the linkage information of objects needs to be stored. This data structure is a 'link info list' in the object info, wherein an element of the list contains the position of objects in the job that are referred from it. As an object is added to the job, the list in the object info of the object containing a reference to it is updated with an entry of the referred object's position in the job. If an object has already been added to the job and is again passed to the argument stream (case when object is referred from more than one object), then only an entry corresponding to the new link is made in the 'link info list' of the referring object's object info. For example, if an object A contains a reference to object B. Both A and B are passed as arguments to job J1. If B gets added to the job before A and A becomes available to the job later, then rather than adding B again (A refers to B), only an entry corresponding to the position of B in the job is made in the 'link info list' in A's object info. Sending new objects to be added to a job before sending the linkage information ensures that position contained in the 'link-info list' of each object is valid in slave process. Referred objects are already created in the slave process and their references in the object can be initialized to the right objects.

In the slave process, after the completion of parallel procedure or after the end of all the operations on the returnable synchronous objects in the slave process, the objects are set as over. **Figure 24** shows the process of transferring an object's ownership after it has been used by the current job. At the outset, it is determined whether the object has a next owner or not (24.1). If there is a next owner, then the object's owner is changed to be the next owner (24.2) and the object is added to the new owner's "new object available list" (24.3). If there is now new owner, then the object's current owner is set as Null (24.4) and the object is added to the "object to be freed list" (24.5). Value of these objects is sent back to the master process if they are available to the slave process. Value of the returnable objects, not available, is sent as they become available to the slave process. On receiving the value of returnable child objects from the slave process, parent objects are updated in the master process.

To return objects back to master objects from the slave process, a similar approach to that in master process is applied. In the slave process, a list of object info 'return object list' is built on the similar lines as the "link info" list in the master process. After object is set as
5 'over', all the returnable objects received from the master process (their references are present in received object list) are added to the list. An object may be set as 'over' with the end of all modification operations on it or after the completion of parallel procedure. A job object corresponding to master process is made as object values are now to be transferred to master process. 'Return object list' is actually an object info list in the
10 master process (job) object that is the last owner of the object. As an object becomes available to the job (ownership of the object is transferred to it), object value is transferred to the master process. Similar to the jobs in master process, a 'new objects available list' is maintained for master process (job) object. The master process object also gets added to 'jobs to be updated list' and is updated similar to jobs in master
15 process. The master process (job) object is always updated in the last.

The process of receiving returnable object list updates in the master process is shown in **Figure 25**. After receiving the object info count (25.1) and determining whether all object info has been received or not (25.2), the object position of all the remaining
20 objects in the is stored in the "returned object" list (25.3). The position of the object is the same as the object's position in the "return object" list in the slave process. The object's value and link info is received (25.4) next, following which the object value is deserialized (25.5). The status of the object is set as "received" (25.6).

25 If a new object created in slave process is made reachable from an object received from master process, the object is created in master process and linked to the appropriate object. All the linkage structure changes made in the slave process is reflected in the master process. Master process receives class info of new objects created in slave process and linkage information (position in object info list when returning objects) to link them
30 from appropriate object in master process.

In addition, **Figure 26** shows the process of receiving of the object values in the master process. In this process new values of the objects are received back (26.4) and objects are updated. The object's association information, i.e. the list of position in the "return object list" of all objects which contain reference to the object, is also received (26.5) and associated with the object. Finally the object's status in the master is set as "received".

Slave process maintains 'new objects available list', 'object info list' and 'objects to be transferred list' as the master process. Operations on these lists are similar to those in jobs in master process. In the slave process, one 'Return object list' of object info of objects containing references to objects that have to be transferred back to the master process. All returnable objects received from master process are added to this list in the slave process.

Distributed systems are prone to network failures and computer shutdowns and crashes. The invention provides automatic fault tolerance, wherein the programmer is relieved from writing code for fault detection and recovery. Fault detection is trivial and can be easily done by Pinging nodes at regular intervals. Master process can also verify if the slave process to which it transferred parallel procedure execution is running as normal or not.

There can be three types of faults: process failure faults, node failure fault or a network failure fault. Process failure is abnormal termination of any of the executing processes. If a process gets abnormally terminated on a node, the master process can throw the job again to another node. Thus a recovery from a 'process failure' is possible by simply rescheduling the process again on the same or any other node. A 'node failure' can be recovered from, by rescheduling each of the processes executing on it to other nodes. A 'complete node failure' can be recovered from, by rescheduling each of the processes executing on it to other node. **Figure 28** demonstrates a 'complete node failure' scenario. In the figure, failure of Node3 occurs while executing Job2 (28.15) assigned by Node1 (node of the master process). On detection of failure of Node3 (28.5), Job2 is rescheduled to run on Node2. Master process maintains complete job object till all the returnable objects are received by it. In case of any failure in the execution of the slave process, the

job can be rescheduled using the job object. The process of rescheduling a slave process is similar to the process of scheduling the process for the first time. Again same parallel procedure ID, list of arguments objects, class info of already added objects, and object value and linkage information of available objects is resent to the slave process (28.10) in the same fashion. Subsequently, the job2 executes on the Node2 (28.12) from the beginning.

As returnable objects are not changed till they do not return (their ownership is with the job till they are returned back to master process), so in the master process their data content remains intact as the master process gets blocked on accessing them. However, non-returnable objects can be modified. Therefore, to support re-execution of parallel procedure, value of objects transferred as non-returnable need to be stored (linkage information is already stored in the object info). Whenever a non-returnable object's value is transferred to the slave process, it is serialized to a buffer. During rescheduling, object's value is transferred directly from the buffer.

A fault may also occur while updating a returnable object on master process as the object may get updated only partially, therefore, returnable objects also need to be serialized to a buffer. Another scenario where returnable objects need to be serialized is when a fault occurs between two object updates on the master process. Few objects may get completely updated and their ownership may be released by the job, while the rest of the objects may still be owned by the job waiting to be updated. After ownership is released by the job, the object(s) in the master process may be modified. However, when rescheduling the process to be run on some other node in the event of some failure, the object's value that needs to be transferred is the one which it had at the point of calling the parallel procedure initially and not the recently modified value.

Rescheduling processes may lead to the problem of double updates of returnable objects in the master process. One update can be from the original process and the other from the rescheduled process. To handle this problem a job maintains information about the status

(‘sent’ or ‘received’) of the object, if the object has been received, object value is discarded.

Storing each object in the buffer may lead to unacceptably high memory usage. To lower the memory usage, if an object is not modified between two calls to parallel procedures, then only a single copy of the buffer may be used by both jobs. The buffer of an object added as non-returnable to more than one slave processes at the same instance can be shared by these processes. However, object buffer of objects added as returnable cannot be shared in this way.

To keep a single buffer of the objects having same value in more than one call to parallel procedures, buffer of each object is kept separate and a common buffer for the complete job is not used. The object info of objects sharing a common buffer refers to this shared object value buffer. Beside the object info, object itself also contains reference to the object value buffer made during last serializing operation on the object. Synchronous object also contains a flag that is ‘true’ only when object value buffer referred by the object contains its current value, that is, object has not been modified after the last serialization operation. Therefore, when any modification is made to the object, the flag is made ‘false’. When object is serialized to be sent as non-returnable, the flag is set ‘true’. However, if object is serialized to be sent as returnable, the flag is set ‘false’ because the object then becomes available to the slave process for modification causing the value in the buffer to be incorrect. If object is sent with the flag set as ‘true’, new buffer is not made and the same buffer is reused by having the object info in the new job to refer to the same buffer. Reference is copied directly from the object value reference in the object. If the flag is set as false, a new copy of the buffer is made and object value references both in object and object info are changed to the new buffer.

For executing the jobs of the programmer’s application over different computers on the network, each computer needs a complete or partial copy of the executable file of the programmer’s application. Before any job is assigned to a computer on the network, the computer should have the executable running as a slave process, waiting for a job to be

assigned to it. A registry service runs on all the computers that participate in the distributed execution of the application. The task of the registry service involves among other things:

- 5 • Registration of programmer's application: The registry service waits for the programmer's application to register by signaling the service on startup.
- Providing the executable file of the application to slave nodes participating in the execution: After registering the application on the master computer, the registry service broadcasts a 'query slave message'. 'Query slave message' is received by the
10 registry service executing on the other nodes on the network. After receiving the query slave message, the registry service of the slave(s) requests for a copy of executable file of the application. The registry service on the master then transfers the requested file to the slave nodes.
- Executing the job after the transfer of executable file: The application running on the
15 slave registers with the registry service on startup and waits for a job to be assigned to them for execution by the master.
- Updating the application at regular intervals with the list of computers executing the programmer's application:

20 It will be apparent to those of ordinary skill in the art that the foregoing description of the invention is merely illustrative and not intended to be exhaustive or limiting. Various modifications can be made within the scope of the mentioned invention for example, a slave process can be executed with arbitrarily higher priority. A parallel procedure that would have been called before a particular procedure in the sequential run of the program
25 may be given higher priority. Master process may number its slave processes executing the parallel procedure using an ID (number) which may be the ID of the master process concatenated with rank of the slave process in the list of parallel procedures called from it. The two IDs can be compared using lexical string comparison and the slave process having the smaller or larger ID, as the case may be, will be executed first.

30

The number of processes that are blocked and waiting to gain ownership of a particular synchronous object presently owned by the process can also determine priority of process. On getting blocked a process may send the object position on which it got blocked so that the master process may increase the priority of the job having the ownership of the object. If a process has yet not received the value of the object from its master process, then the number of processes blocked on the object builds up at each level of master – slave hierarchy. This build up continues to the process in which object has originally been created and is sent to the process having its ownership whose priority is increased accordingly.

10

Figure 29 depicts a pictorial representation of a network of data processing systems in which the present invention may be implemented. Network data processing system contains a network 29.1, which is the medium used to provide communications links between various nodes connected together within network data processing system.

15 Network 29.1 may include connections, such as wire, wireless communication links, or fiber optic cables.

In the depicted example, a node executing the master process 29.2 is connected to network 29.1. In addition, nodes 29.3, 29.4 and 29.5 are also connected to network 29.1.

20 These nodes 29.3, 29.4 and 29.5 may be, for example, personal computers or network computers. In the depicted example, the node executing the master process 29.2 creates the jobs and sends them to the slaves 29.3 and 29.4 for execution. The slave nodes 29.3 and 29.4 may further transfer the execution of the jobs to some other slave nodes such as 29.5, thereby becoming the master for that particular slave node 29.5.

25

The mechanism of the present invention allows for the updation of data arguments sent as returnable to the slave on the master node that sent them to the slave.

30 Network data processing system 29.1 may include additional computing systems and other devices not shown in the depicted example. In the depicted example, network data processing system 29.1 is a typical Local Area Network with network 29.1 representing a

collection of networks and gateways that use the TCP/IP suite of protocols to communicate with one another. Of course, network data processing system may also be implemented as a number of different types of networks, such as for example, an intranet or a wide area network (WAN). Figure 1 is intended as an example, and not as an architectural limitation for the present invention.

Figure 30 is a block diagram that illustrates a typical device incorporating the invention. The device (30.1) consists of various subsystems interconnected with the help of a system bus (30.2). Each device (30.1) incorporates a storage device (30.5) that is used to store the sequential program and means for executing it in parallel.

Those of ordinary skill in the art will appreciate that the means to execute the program are instructions for operating on the nodes of the system. The means are capable of existing in an embedded form within the hardware of the node or may be embodied on various computer readable media. The computer readable media may take the form of coded formats that are decoded for actual use in a particular information processing system. Computer program means or a computer program in the present context mean any expression, in any language, code, or notation, of a set of instructions intended to cause a system having information processing capability to perform the particular function either directly or after performing either or both of the following:

- a) conversion to another language, code or notation
- b) reproduction in a different material form.

At the computer system executing the master process, the keyboard (30.10), mouse (30.11) and other input devices connected to the computer system through the I/O interface (30.9) are used to input the sequential program and specify the parts of the program to be executed in parallel as "parallel procedures". Following this the program is executed and the instructions encoded in the means to execute the program in parallel are transferred from the storage device (30.5) to the memory (30.4), which holds the current instructions to be executed by the processor (30.3) along with their results, through the internal communication bus (30.2). The processor (30.3) executes the instructions by

fetching them from the memory (30.4) to create jobs for each call to a parallel procedure. The computer system uses the networking interface (30.8) to send the jobs to the target nodes over a network such as the LAN (30.12).

- 5 At the slave node, the job is received over the LAN (30.12) through the networking interface (30.8) and the processor (30.3) executes the job.

Those of ordinary skill in the art will appreciate that the hardware depicted in Figure 30 may vary. For example, other peripheral devices, such as optical disk drives and the like,
10 also may be used in addition to or in place of the hardware depicted. The depicted example is not meant to imply architectural limitations with respect to the present invention.

Similar to arguments sent to a procedure, a procedure's return value can also be
15 synchronized. Return value of a parallel procedure can be implemented on similar lines as of returnable objects wherein, a parallel procedure can only return a reference to a synchronous object. Return object can be specified to the job by sending them to the job in the same manner as a returnable object is sent. On completion of the parallel procedure the return object can be sent to the master process.

20

The system allows executing a program sequentially for the purpose of debugging or otherwise by calling the parallel procedures directly.

In the master process all the communication is non-blocking wherein, neither master
25 process nor the slave process gets blocked during communication. Neither of them get involved in the actual process of data communication as the framework provides for separate threads for the purpose of data communication. All jobs are updated for synchronization purposes before any data transfer starts.

30 If more than one job is to be sent same arguments, the data can be multicast to them saving valuable time in data communication. Parallel procedures can group together by

adding job objects to a job group object that allows them to be called together. The framework subdivides the group into some finite parts depending on the resources available and transfers them to processing nodes. Each subdivided group is further divided till only one parallel procedure remains in each group. In this process, the load of distributing parallel procedures also gets distributed and execution efficiency is improved as the value of objects common to the parallel procedures in a group is transferred once only during the transfer of the process group.

The computing environment also provides for interface to its administrator through a Process viewer. The Process viewer is an application started by the registry service at startup and is responsible for showing the state of execution of the programmer's application at any moment of time. It shows all the information about the nodes involved in the distributed processing. The information may include its hardware information or resource availability on the each computing node. The process viewer can also be used for getting the information about each process running on the system along with its ID, its priority, state (ready or suspended), its execution time etc. The process viewer can also be used to control the distribution of processes and to suspend processes or forcing process distribution to a particular node or set of nodes.

It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the present invention is capable of being distributed in the form of computer instructions forming a computer program in which these instructions may be embodied on various computer readable media. The present invention applies equally regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media include recordable-type media, such as a floppy disk, a hard disk drive, a RAM, CD-ROMs, DVD-ROMs, and transmission-type media, such as digital and analog communications links, wired or wireless communications links using transmission forms, such as, for example, radio frequency and light wave transmissions. The computer readable media may take the form of coded formats that are decoded for actual use in a particular data processing system.

We Claim:

1. A method for executing a sequential program in parallel with automatic fault
5 tolerance on a computer network comprising the steps of :
 - specifying each part of the sequential program to be executed in parallel as a parallel procedure,
 - executing the sequential program as a master process,
 - 10 – creating a job corresponding to each call to a parallel procedure in the master process,
 - adding objects, that are instances of an in built wrapper class, as arguments to said job, and
 - sending the job along with the added arguments to a slave node and executing
15 said job as a slave process on the node.
2. The method as claimed in claim 1 further comprising the step of providing automatic fault tolerance in the event of slave process, slave node or network
20 failure.
3. The method as claimed in claim 1 further comprising the step of providing an interface for an administrator to view all information regarding slave nodes of the network involved in the execution of the program and to control the distribution of
25 processes.
4. The method as claimed in claim 1 wherein said slave process can further transfer the execution of at least one job received from its master to another slave node and becomes the master process for this new slave process.
- 30 5. The method as claimed in claim 1 wherein the step of specifying a parallel procedure comprises the steps of :
 - specifying the ID of the parallel procedure,

- specifying objects to be added as arguments to said parallel procedure, and
 - specifying type of each said argument added to the parallel procedure.
- 5 6. The method as claimed in claim 5 wherein the ID of the parallel procedure includes the name of the procedure, address of the procedure, an instance of a class containing the parallel procedure or bound with the procedure and a symbol that is bound to the parallel procedure.
- 10 7. The method as claimed in claim 1 wherein the special in built wrapper class is the “SynchronousObject” class that is provided by the framework to encapsulate data synchronization features for each regular data type of the language platform.
- 15 8. The method as claimed in claim 7 wherein a wrapper class can be defined for custom data types used by the programmer by deriving it from a “SynchronousObjectBase” as the base class.
- 20 9. The method as claimed in claim 1 wherein said object has an ownership lock associated with it to ensure that only the job having object’s ownership rights at any instance of time is able to modify it.
10. The method as claimed in claim 9 wherein a process other than the process owning the job at a particular instance gets blocked while trying to modify or access the object.
- 25 11. The method as claimed in claim 10 wherein said object has an “ownership queue” associated with it to store references of jobs waiting to get the ownership of the object.
- 30 12. The method as claimed in claim 1 wherein said object passed as an argument to a procedure can refer to other synchronous objects and this linkage structure is maintained for each object.

13. The method as claimed in claim 12 wherein changes made to the references of said object are reflected in its corresponding linkage structure.
- 5 14. The method as claimed in claim 5 wherein type of the added argument can be "returnable" and "non returnable".
- 10 15. The method as claimed in claim 5 wherein an object reachable from an object that has been added to a job as "returnable" is also added to the job as "returnable" while, an object that is reachable from an object added as "non-returnable" is added as "non-returnable".
- 15 16. The method as claimed in claim 5 wherein an object can be added to more than one job as "non-returnable" and its value is transferred simultaneously to all these jobs until a job in which the object is added as "returnable" is encountered or the ownership queue becomes empty.
- 20 17. The method as claimed in claim 5 wherein objects referred to by another object that is not available to the job are not added to the job until the referring object becomes available.
- 25 18. The method as claimed in claim 1 wherein the step of sending the job to the chosen slave node comprises the steps of :
- sending the job info header of the job,
 - sending the "class info" of each argument added to said job,
 - sending object information of each said argument and
 - sending values of said arguments to the slave node as and when they become available.

30

19. The method as claimed in claim 18 wherein the job info header includes Job ID identifying the job, the parallel procedure ID and the number of objects added to it as arguments.
- 5 20. The method as claimed in claim 18 wherein every synchronous object has a “serialize” procedure associated with it to send its value to the slave or the master process by copying it to a data stream.
- 10 21. The method as claimed in claim 20 wherein every synchronous object has a “deserialize” procedure associated with it to receive and update its value on a slave or master process by copying data put on the data stream using the “serialize” procedure to the object variables.
- 15 22. The method as claimed in claim 18 wherein if the object is passed as “returnable” to a slave process, it becomes unavailable to the master process, i.e., object’s ownership is transferred to the slave process executing the job.
- 20 23. The method as claimed in claim 22 wherein an argument passed as “returnable” to a slave process is updated at the master as soon as the last instruction modifying the object is executed and the ownership is transferred back to the master process.
- 25 24. The method as claimed in claim 23 wherein the last instruction modifying the object is determined either by the compiler or by the programmer specifying it explicitly or by the end of the procedure execution.
- 25 25. The method as claimed in claim 18 wherein if the object is passed as “non-returnable” to a slave process, then the object remains available to the master process, i.e., the ownership does not get transferred.

26. The method as claimed in claim 1 wherein a child synchronous object, on which all operations are performed at the slave side, is created at the slave node for every argument sent to it by the master process using the object's "class info".
- 5 27. The method as claimed in claim 2 wherein in case of a node failure all processes executing on that node are rescheduled to run again on a different node whereas, in case of a process or network failure all processes are rescheduled to run again on same or a different node.
- 10 28. The method as claimed in claim 27 wherein the status of an object is maintained as "sent" or "received" in a job to avoid updating an object more than once during process rescheduling.
- 15 29. The method as claimed in claim 1 comprising the step of grouping the various jobs in the form of a job group for sending to a node so as to save communication overhead.
- 20 30. The method as claimed in claim 1 wherein each slave process can be assigned a priority by the master process and the master process can modify this priority as required.
- 25 31. The method as claimed in claim 1 wherein the priority of a slave process can be based on different criteria including order in which the procedure executed by the process occurs in the sequential program and the number of processes blocked while trying to access an object owned by the job being executed by the process.
- 30 32. A computing system enabled for executing a sequential process in parallel with automatic fault tolerance capability comprising of :
- means for specifying each part of the sequential process to be executed in parallel as a parallel procedure,

- means for executing the sequential process as a master process,
- means for creating a job corresponding to each call to a parallel process in the master process,
- means for adding objects, that are instances of an in built wrapper class, as arguments to said job, and
- means for sending the job along with the added arguments to a slave node and executing said job as a slave process on the node.

33. The system as claimed in claim 32 further comprising means for providing automatic fault tolerance in the event of slave process, slave node or network failure.

34. The system as claimed in claim 32 further comprising of interface means for an administrator to view all information regarding slave nodes of the network involved in the execution of the program and to control the distribution of processes.

35. The system as claimed in claim 32 wherein means for specifying a parallel procedure comprise of :

- means for specifying the ID of the parallel procedure,
- means for specifying objects to be added as arguments to said parallel procedure, and
- means for specifying type of each said argument added to the parallel procedure.

36. The system as claimed in claim 32 wherein means for providing data synchronisation features is a special in built wrapper class, "SynchronousObject".

37. The system as claimed in claim 32 wherein said object has ownership lock means associated with it to ensure that only the job having object's ownership rights at any instance of time is able to modify it.

38. The system as claimed in claim 37 wherein said ownership lock means include semaphores and other synchronisation objects provided by the operating system.
- 5 39. The system as claimed in claim 38 wherein said object has means to store references of jobs waiting to get the ownership of the object not owned by them, after getting blocked while trying to access or modify it.
- 10 40. The system as claimed in claim 39 wherein said means to store references is an "ownership queue" implemented as a linked list.
41. The system as claimed in claim 32 comprising of linkage structure means for each said object.
- 15 42. The method as claimed in claim 41 wherein said linkage structure means are used for maintaining references of objects referred to by an object passed as an argument to a procedure.
- 20 43. The system as claimed in claim 32 wherein means for sending the job to the chosen slave node comprises:
- means for sending the job info header of the job,
 - means for sending the "class info" of each argument added to said job,
 - means for sending object information of each said argument and
 - 25 - means for sending values of said arguments to the slave node as and when they become available.
- 30 44. The system as claimed in claim 43 wherein the job info header includes Job ID identifying the job, the parallel procedure ID and the number of objects added to it as arguments.

45. The system as claimed in claim 43 wherein every synchronous object has means to send its value to the slave or the master process by copying it to a data stream.

46. The system as claimed in claim 45 wherein every synchronous object has means to receive and update its value on a slave or master process by copying data put on the data stream.

47. The system as claimed in claim 33 including means for rescheduling all processes executing on node to run again on a different node in case of a node failure while, rescheduling all processes are rescheduled to run again on same or a different node in case of a process or network failure.

48. The system as claimed in claim 47 including means for maintaining the status of an object as "sent" or "received" in a job to avoid updating an object more than once during process rescheduling.

49. The system as claimed in claim 32 including means for grouping various jobs in the form of a job group for sending to a node so as to save communication overhead.

50. The system as claimed in claim 32 including means for each slave process to be assigned a priority by the master process and allowing the master process to modify this priority as required.

51. The computing system as claimed in claim 32 wherein said computing system comprises of :

- a system bus,
- a communication unit connected to said system bus,
- a memory including a set of instruction connected to system bus, and
- a control unit executing instructions in the memory for carrying out the operations described above.

52. The system as claimed in claim 51 wherein said computing system is connected to other similar systems through a network.

53. The system as claimed in claim 52 wherein said network is any known computing network using any known communication protocol.

54. The system as claimed in claim 52 wherein each said computing system on the said network comprises means to transfer execution of jobs to other computing systems connected to said network.

55. A computer program product comprising computer readable program code stored on a computer readable storage medium embodied therein for executing a sequential process in parallel with automatic fault tolerance capability comprising of :

- computer readable program code means configured for specifying each part of the sequential process to be executed in parallel as a parallel procedure,
- computer readable program code means configured for executing the sequential process as a master process,
- computer readable program code means configured for creating a job corresponding to each call to a parallel process in the master process,
- computer readable program code means configured for adding objects, that are instances of an in built wrapper class, as arguments to said job, and
- computer readable program code means configured for sending the job along with the added arguments to a slave node and executing said job as a slave process on the node.

56. The computer program product as claimed in claim 57 further comprising computer readable program code means configured for providing automatic fault tolerance in the event of slave process, slave node or network failure.

57. The computer program product as claimed in claim 57 further comprising computer readable program code interface means configured for an administrator to view all information regarding slave nodes of the network involved in the execution of the program and to control the distribution of processes.

5

58. The computer program product as claimed in claim 57 comprising of computer readable program code means configured for specifying a parallel procedure, said program code includes instructions configured for:

10

- specifying the ID of the parallel procedure,
- specifying objects to be added as arguments to said parallel procedure, and
- specifying type of each said argument added to the parallel procedure.

15

59. The computer program product as claimed in claim 57 comprising of computer readable program code means configured for providing data synchronisation features is a special in built wrapper class "SynchronousObject".

20

60. The computer program product as claimed in claim 57 wherein said object has computer readable program code means configured to provide ownership lock to ensure that only the job having object's ownership rights at any instance of time is able to modify it.

25

61. The computer program product as claimed in claim 60 wherein said ownership lock means include semaphores and other synchronisation objects provided by the operating system.

30

62. The computer program product as claimed in claim 61 wherein said object has computer readable program code means configured to store references of jobs waiting to get the ownership of the object not owned by them, after getting blocked while trying to access or modify it.

63. The computer program product as claimed in claim 57 comprising of computer readable program code means configured for sending the job to the chosen slave node, said program code includes instructions configured for:

- sending the job info header of the job,
- sending the "class info" of each argument added to said job,
- sending object information of each said argument and
- sending values of said arguments to the slave node as and when they become available.

64. The computer program product as claimed in claim 63 wherein the job info header includes Job ID identifying the job, the parallel procedure ID and the number of objects added to it as arguments.

65. The computer program product as claimed in claim 63 wherein every synchronous object has computer readable program code means configured to send its value to the slave or the master process by copying it to a data stream.

66. The computer program product as claimed in claim 65 wherein every synchronous object has computer readable program code means configured to receive and update its value on a slave or master process by copying data put on the data stream.

67. The computer program product as claimed in claim 58 comprising of computer readable program code means configured for rescheduling all processes executing on node to run again on a different node in case of a node failure while, rescheduling all processes are rescheduled to run again on same or a different node in case of a process or network failure.

68. The computer readable program product as claimed in claim 67 comprising of computer readable program code means configured for maintaining the status of an

object as "sent" or "received" in a job to avoid updating an object more than once during process rescheduling.

5 69. The computer readable program product as claimed in claim 57 comprising of computer readable program code means configured for grouping various jobs in the form of a job group for sending to a node so as to save communication overhead.

10 70. The computer readable program product as claimed in claim 57 comprising of computer readable program code means configured for each slave process to be assigned a priority by the master process and allowing the master process to modify this priority as required.

15 71. A method for executing a sequential program in parallel with automatic fault tolerance on a computer network as described herein with reference to the accompanying drawings.

20 72. A system for executing a sequential program in parallel with automatic fault tolerance on a computer network as described herein with reference to the accompanying drawings.

25 73. A computer program product comprising computer readable program code stored on a computer readable storage medium embodied therein for executing a sequential process in parallel with automatic fault tolerance capability as described herein with reference to the accompanying drawings.

Dated this 29th. Day of August, 2002



Of Anand and Anand, Advocates
Attorney for the applicants

29 AUG 2002

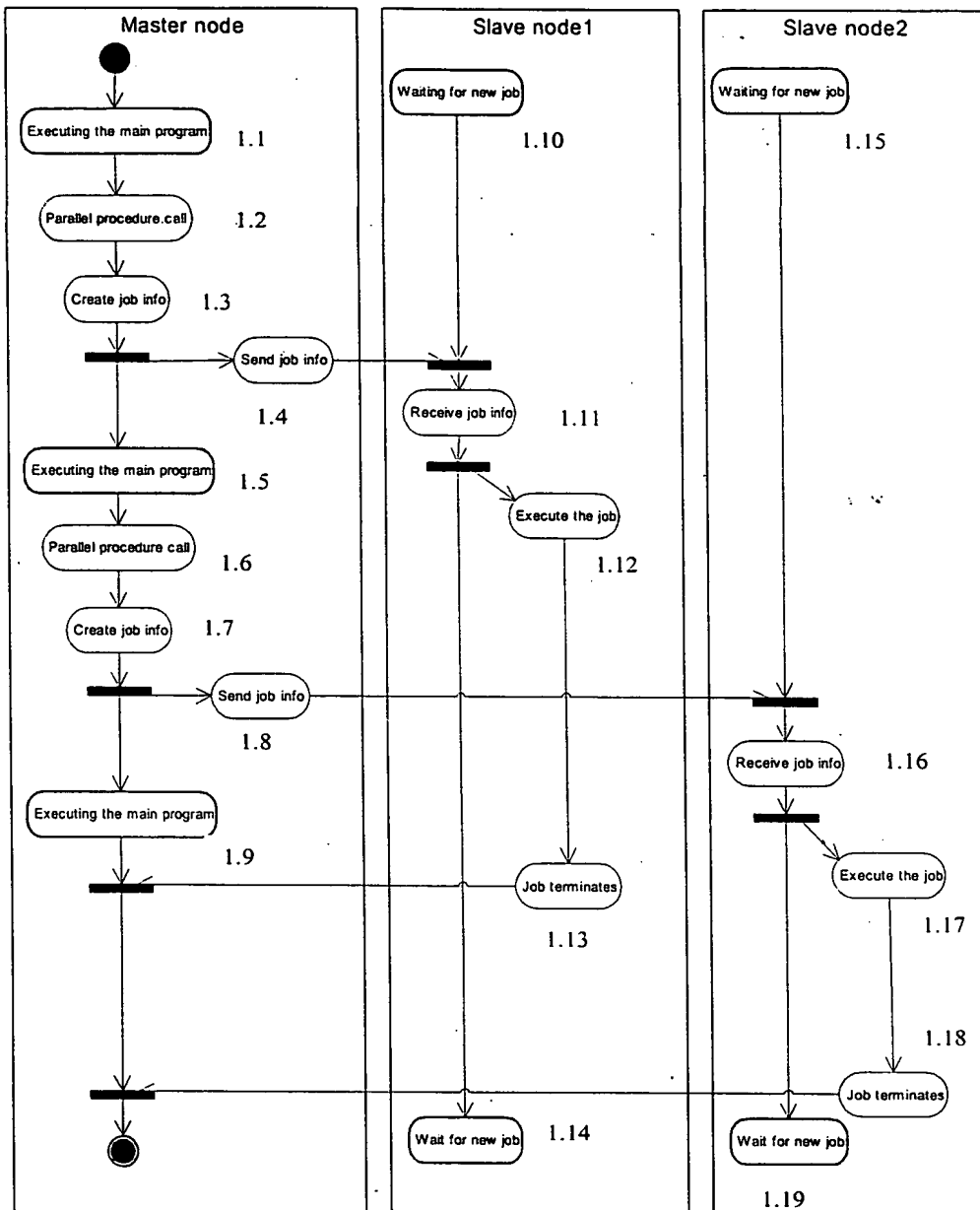


FIGURE 1

Anand and Anand, Advocates,
Attorney for the Applicants

0884-2

29 AUG 2002

DUPLICATE

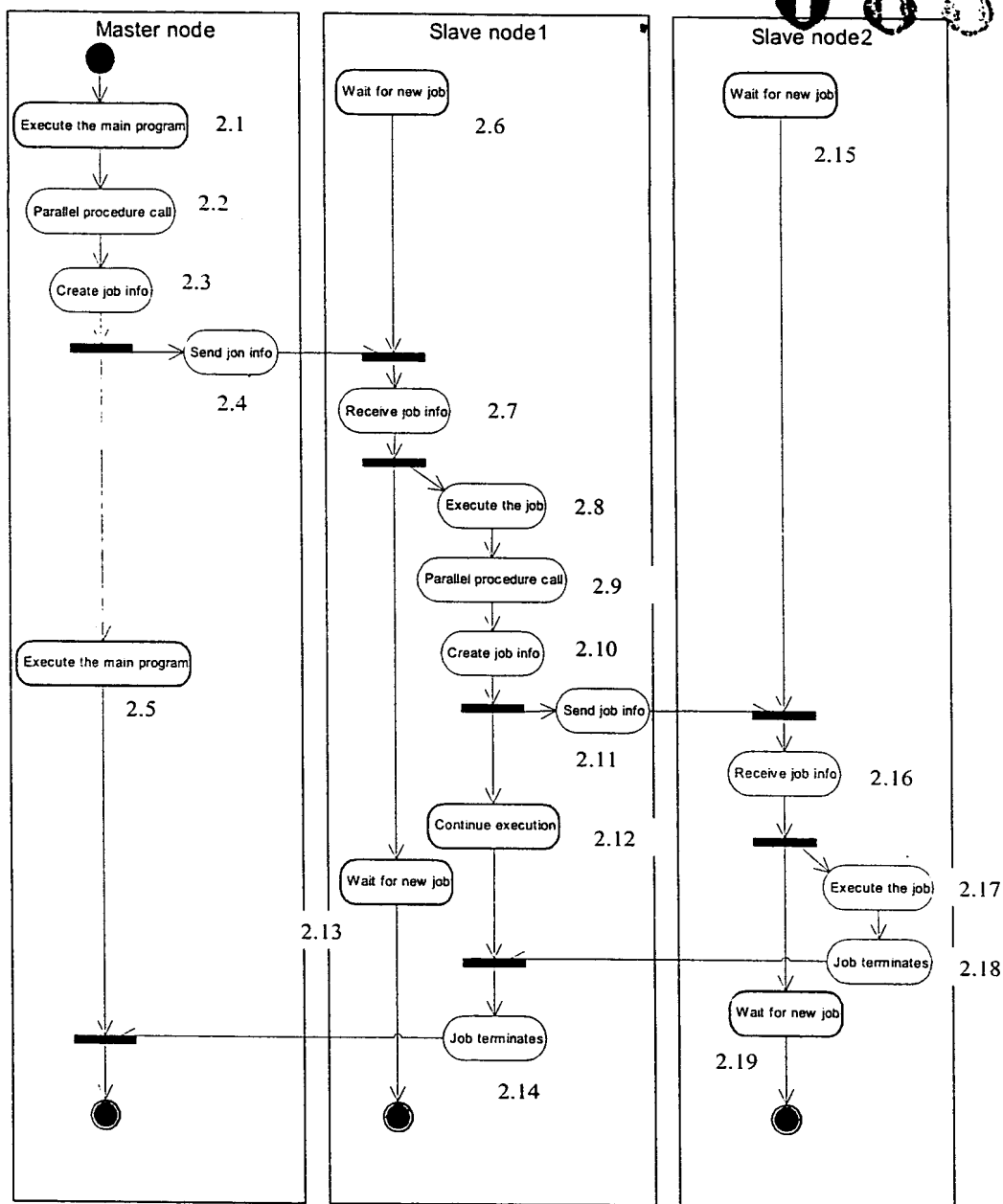


FIGURE 2

Applicant: IIIT, Allahabad
Applⁿ No.:

Sheets: 27
Sheet:

0884-2

29 AUG 2002

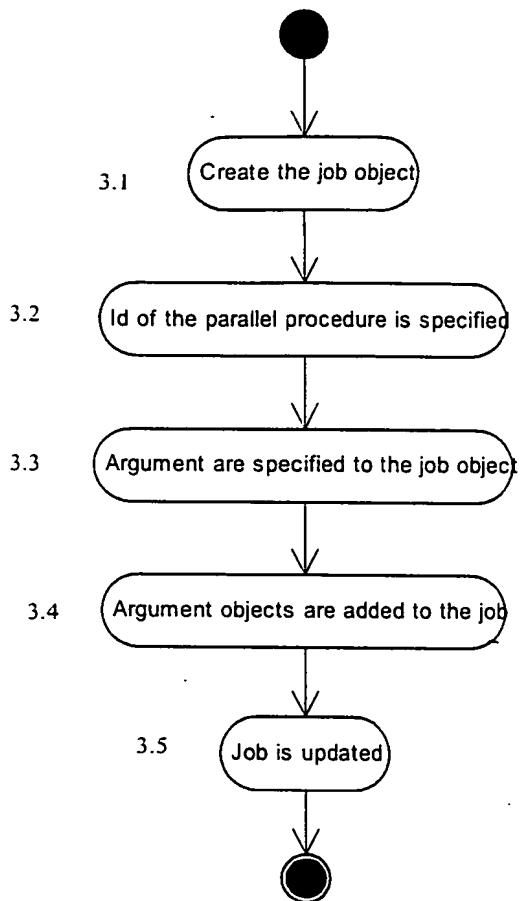



FIGURE 3

DUPLICATE


Anand and Anand, Advocates,
Attorney for the Applicants

29 AUG 2002

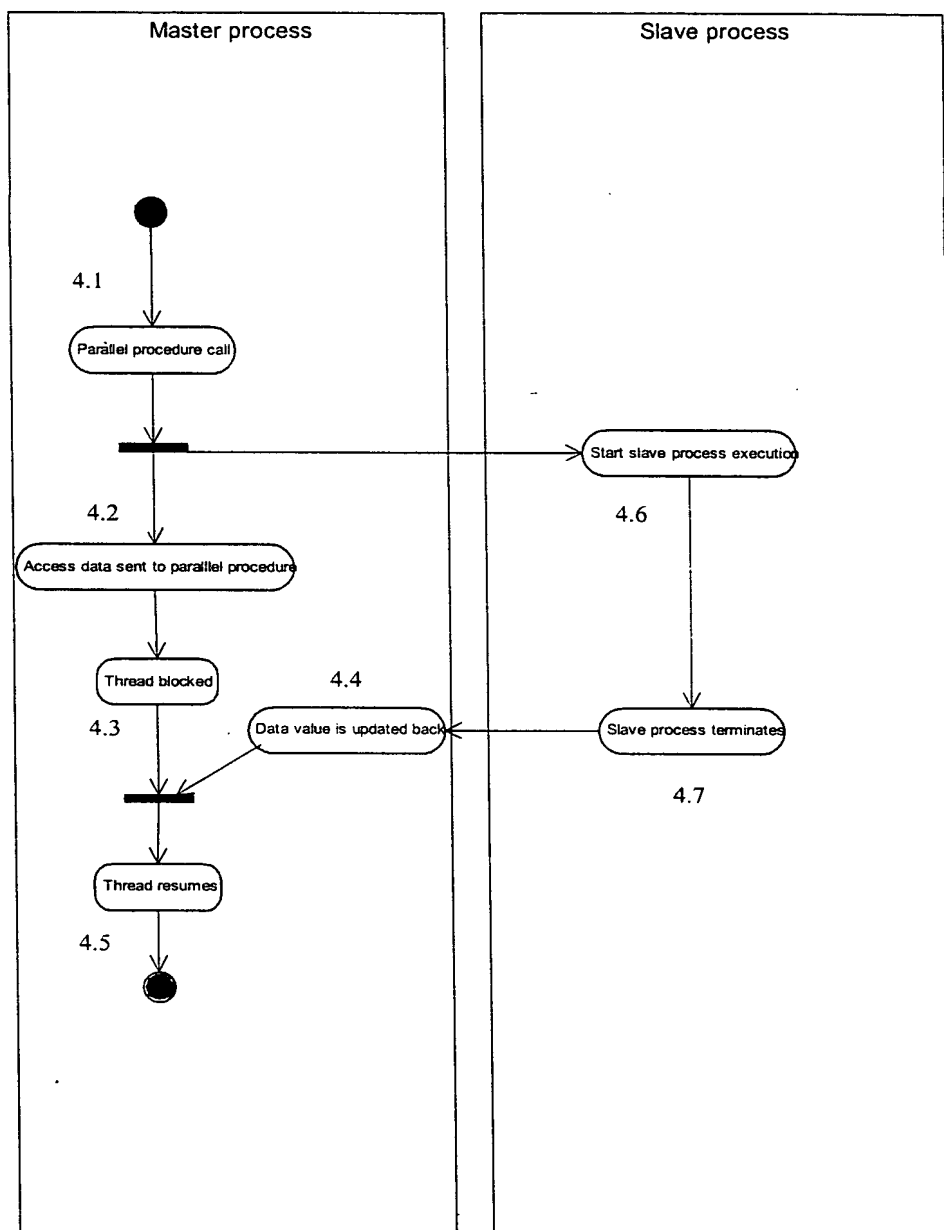


FIGURE 4

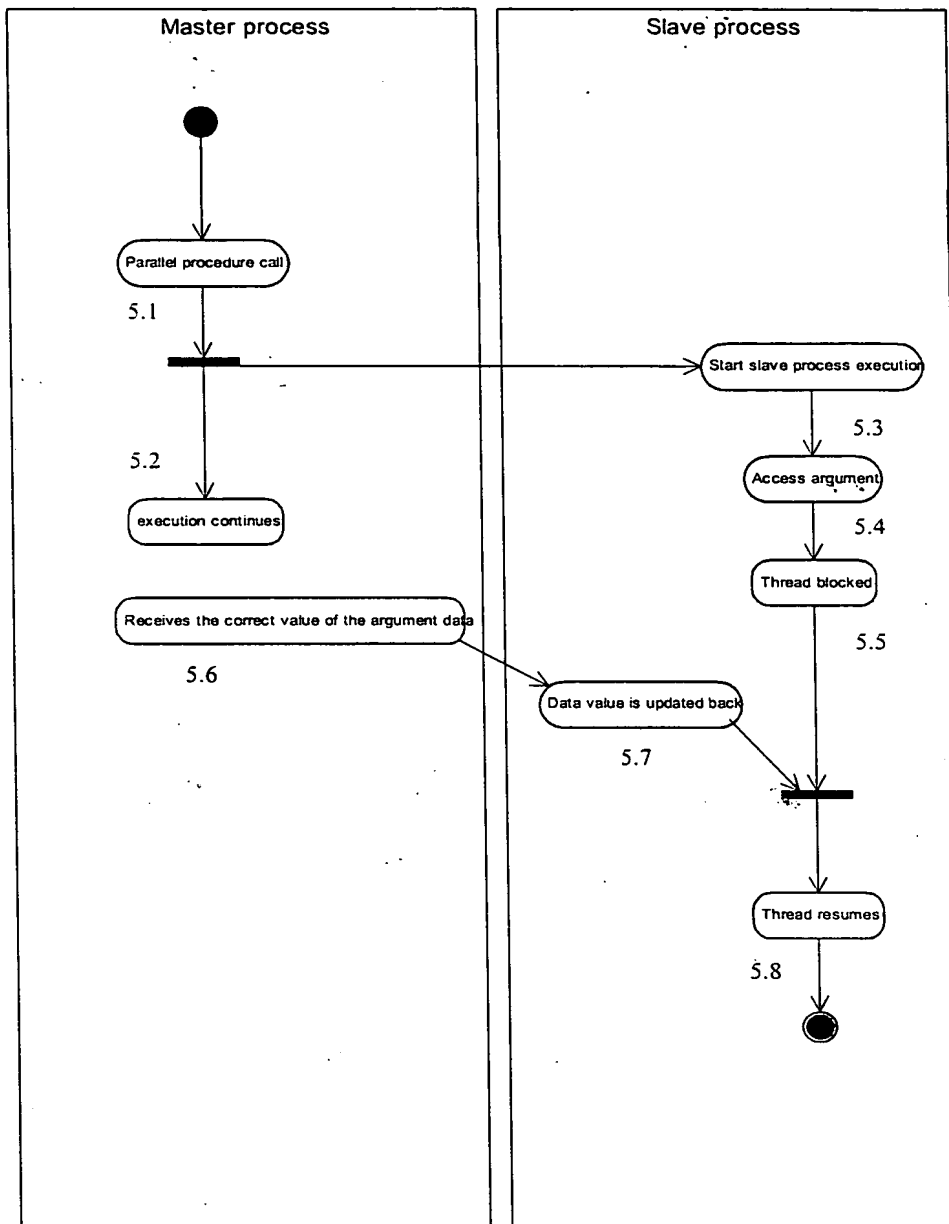


FIGURE 5

Anand and Anand, Advocates,
Attorney for the Applicants

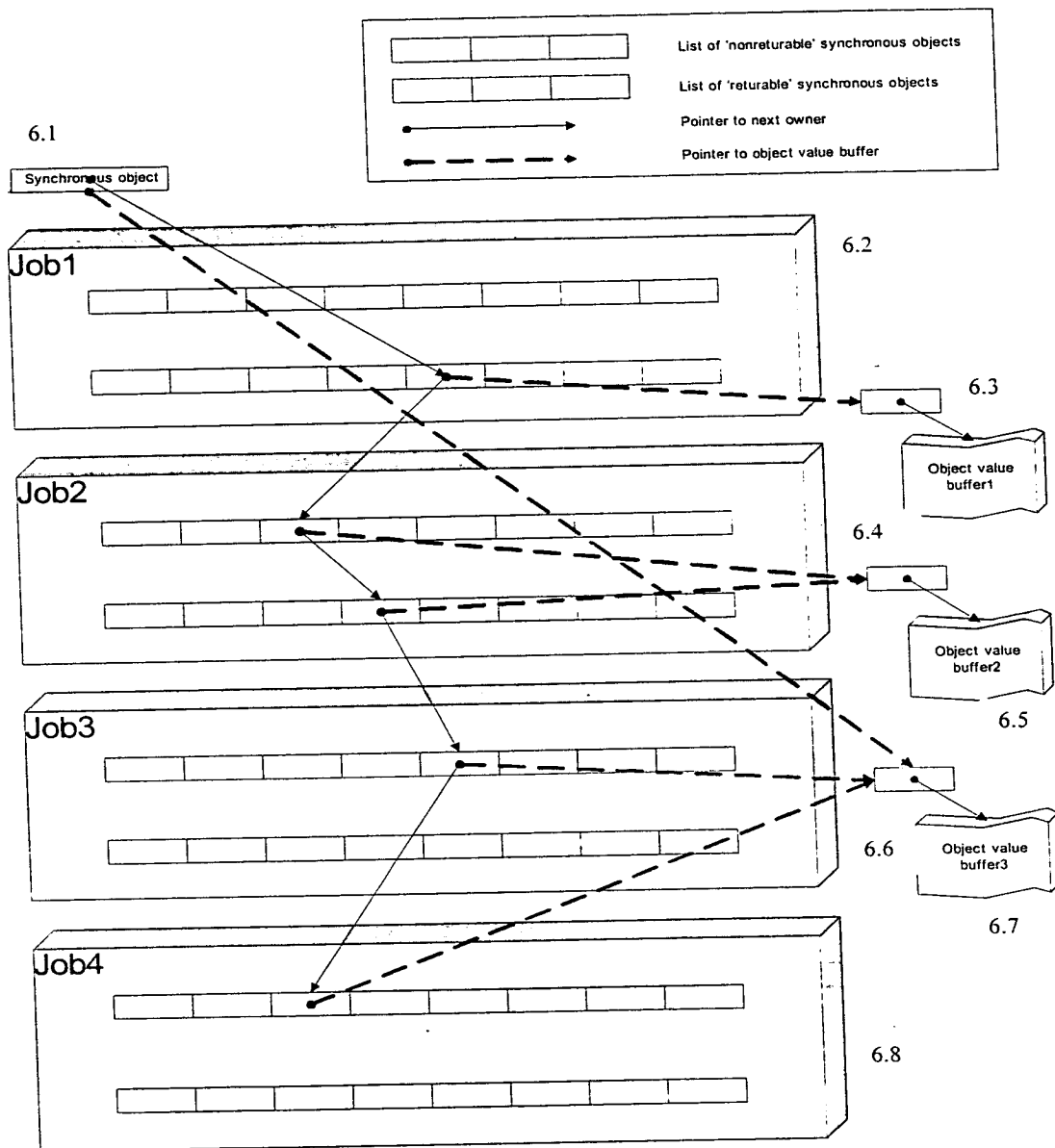


FIGURE 6

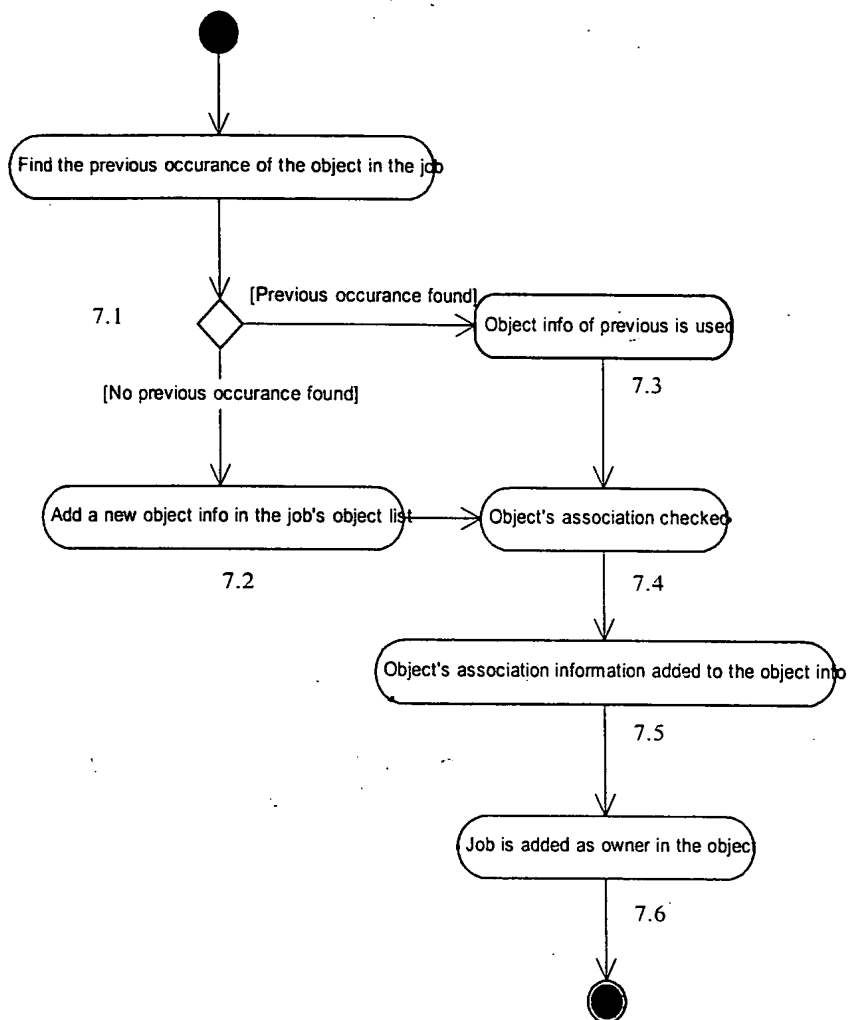


FIGURE 7

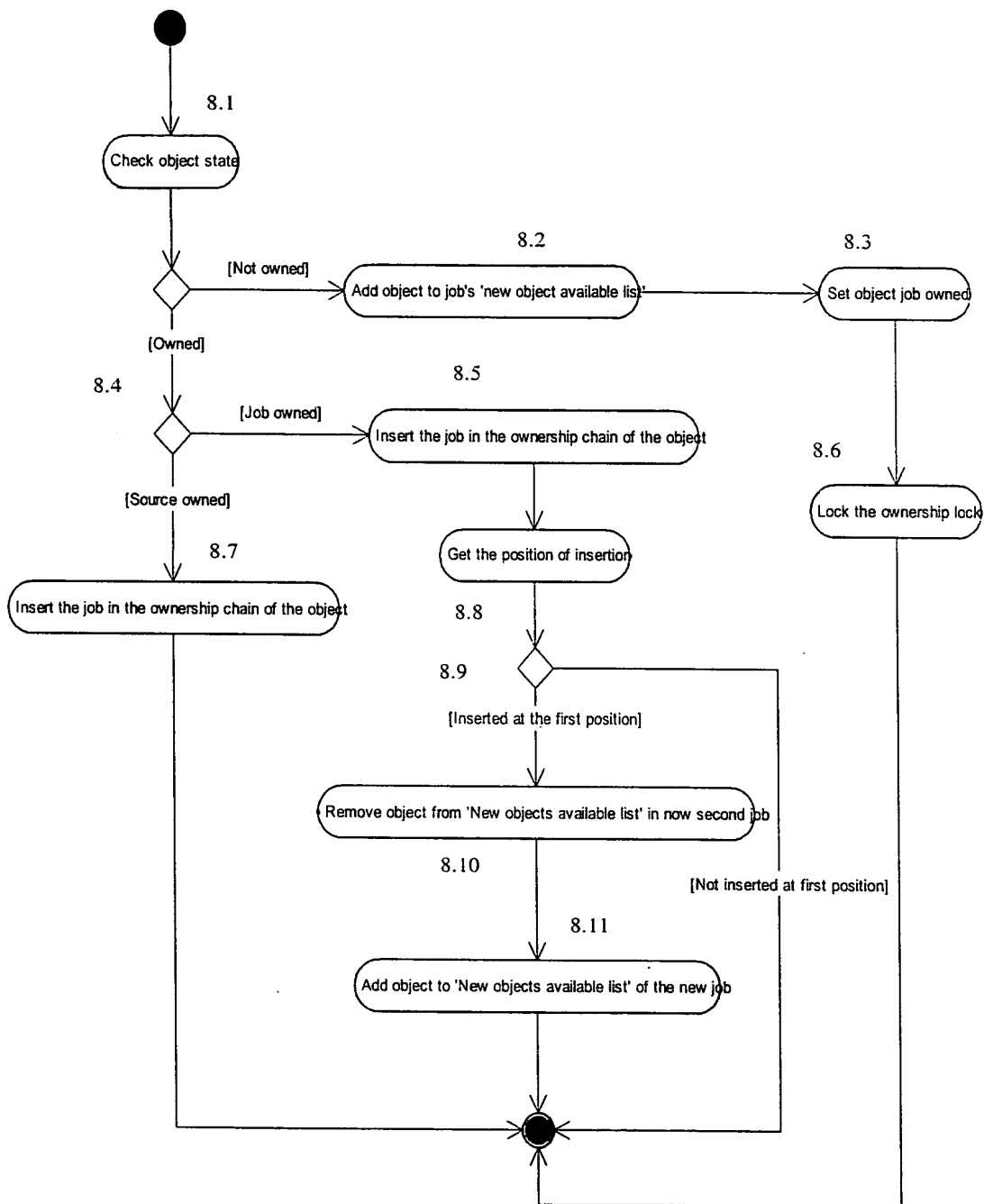


FIGURE 8

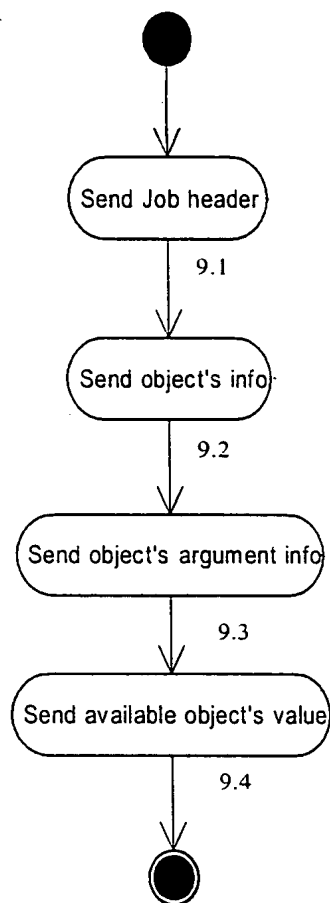


FIGURE 9

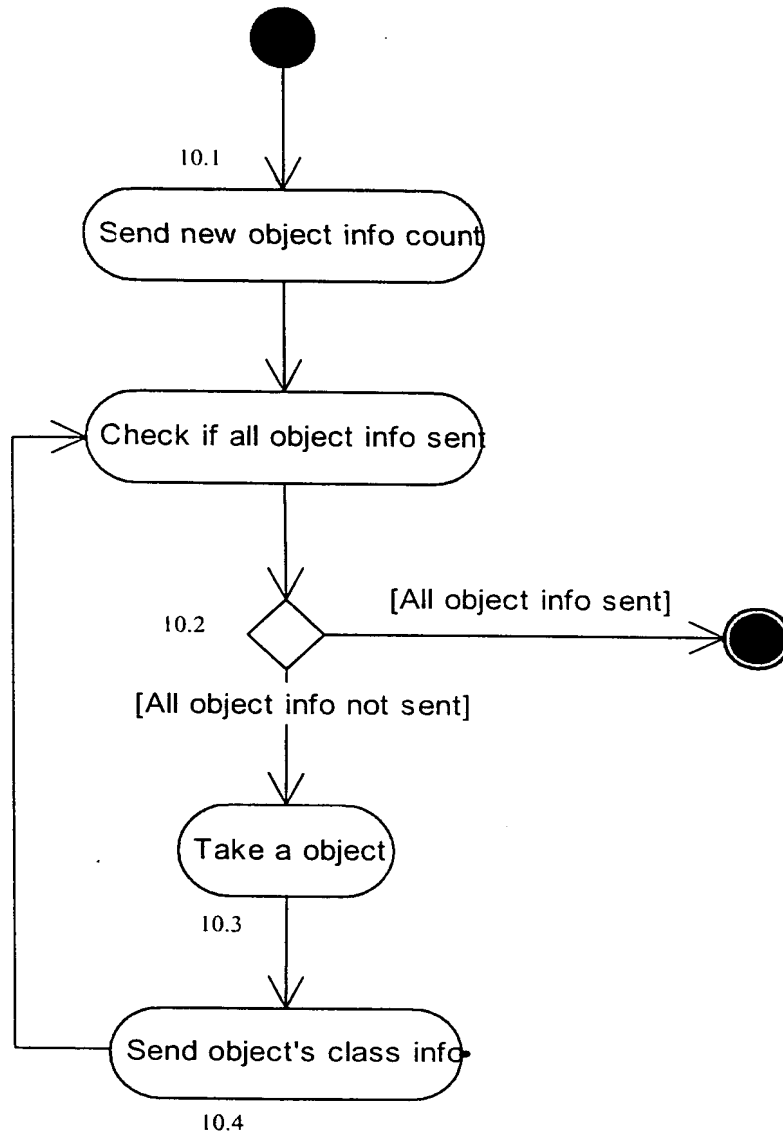
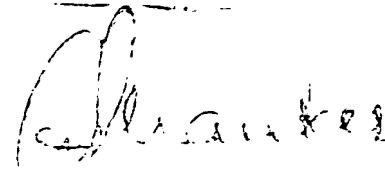


FIGURE 10


Anand and Anand, Advocates,
Attorney for the Applicants

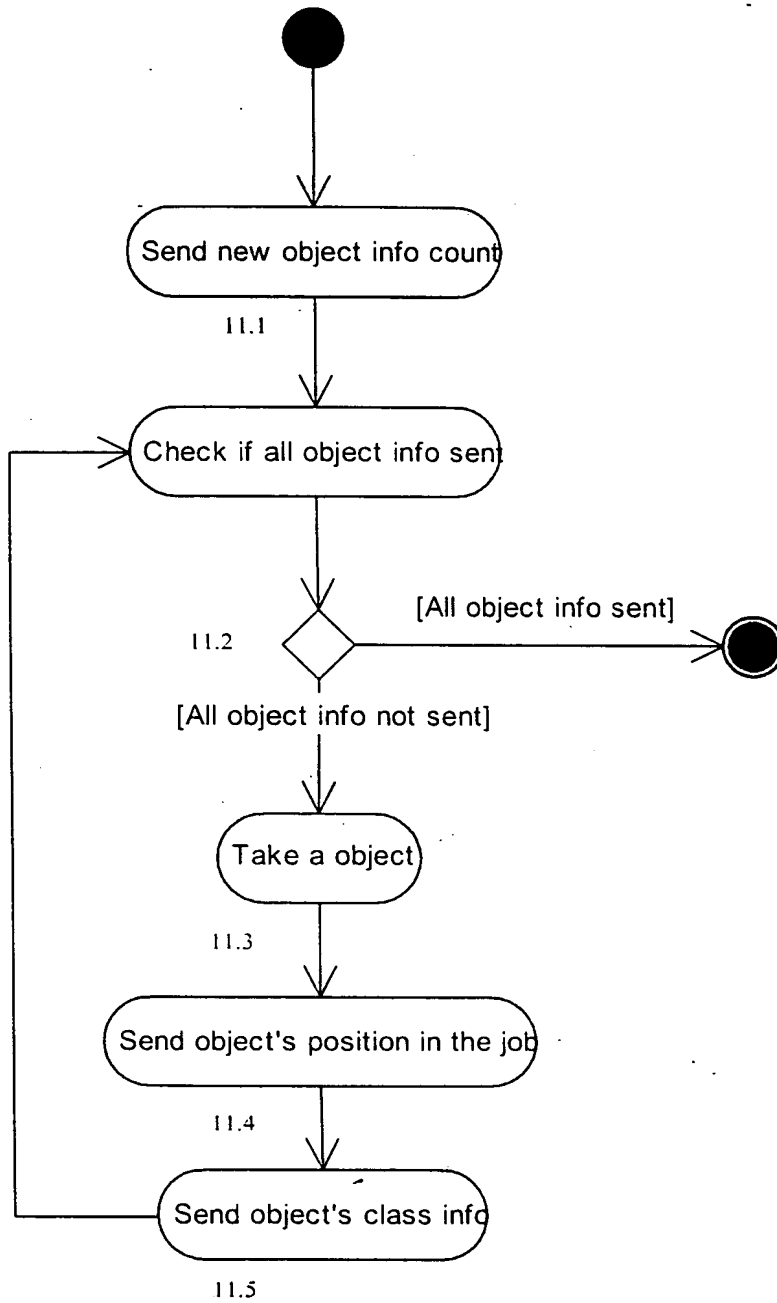


FIGURE 11

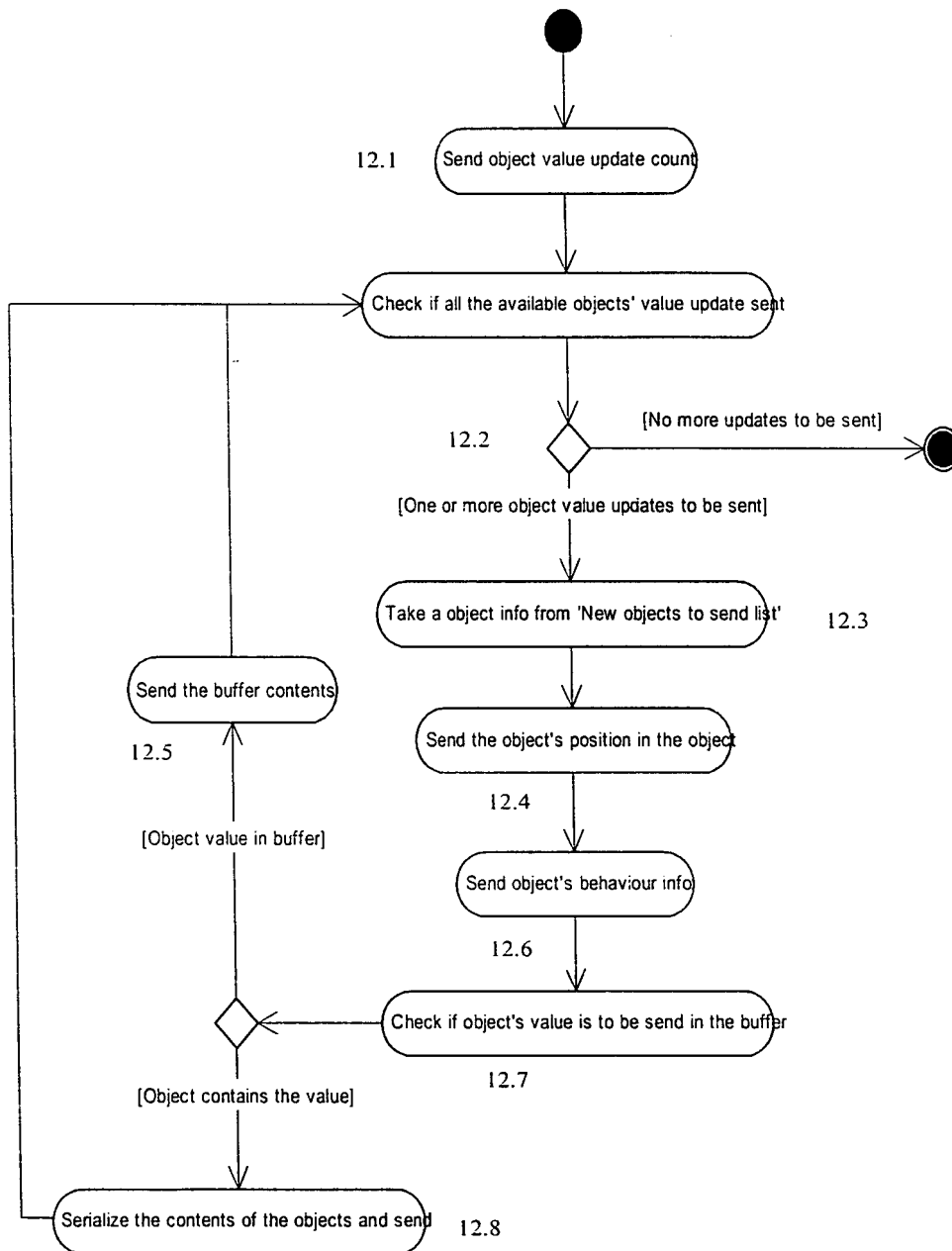
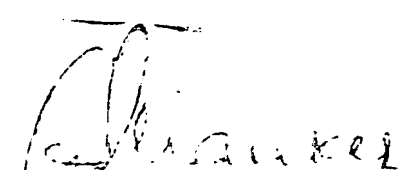


FIGURE 12


Anand and Anand, Advocates,
Attorney for the Applicants

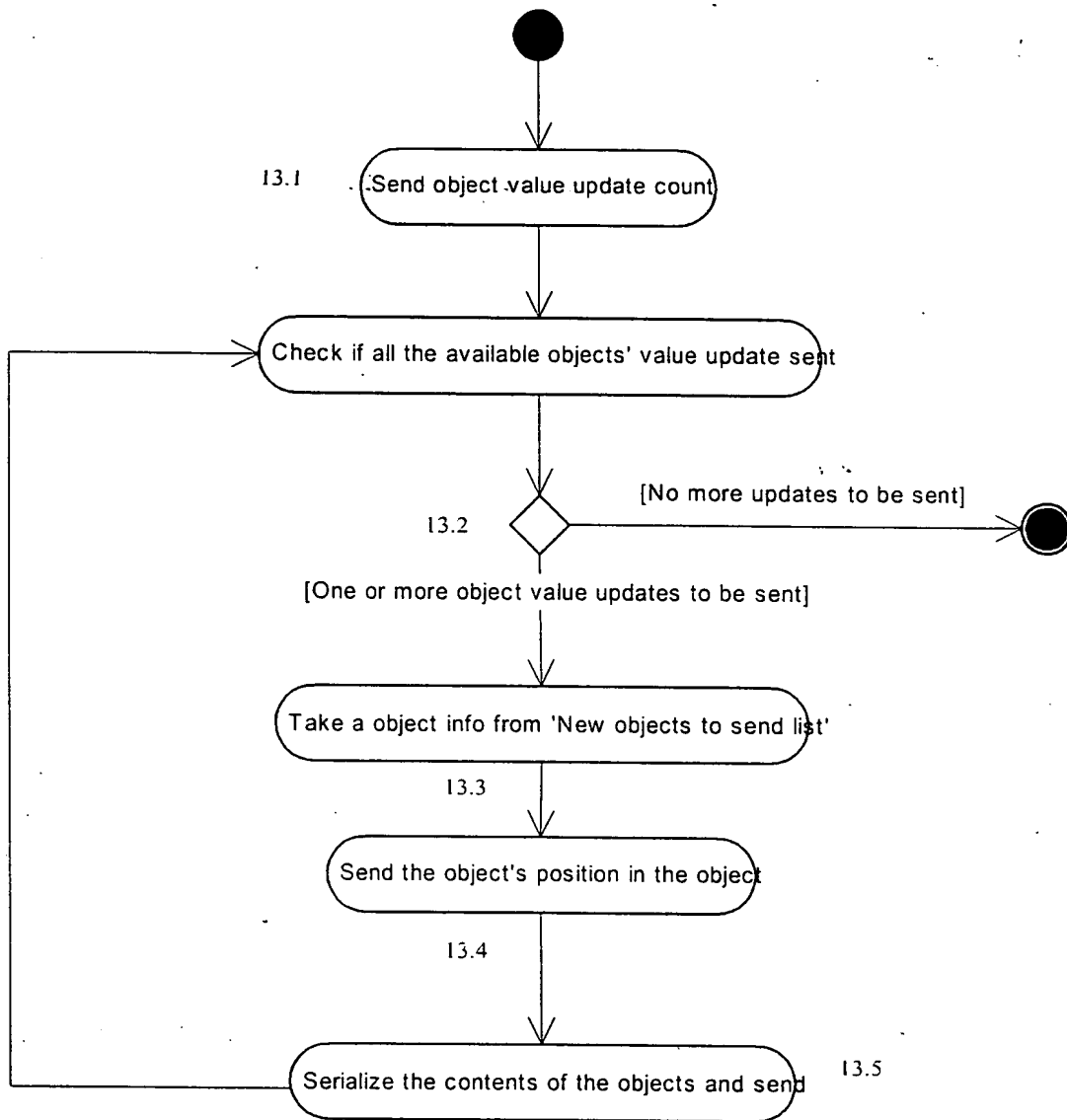


FIGURE 13

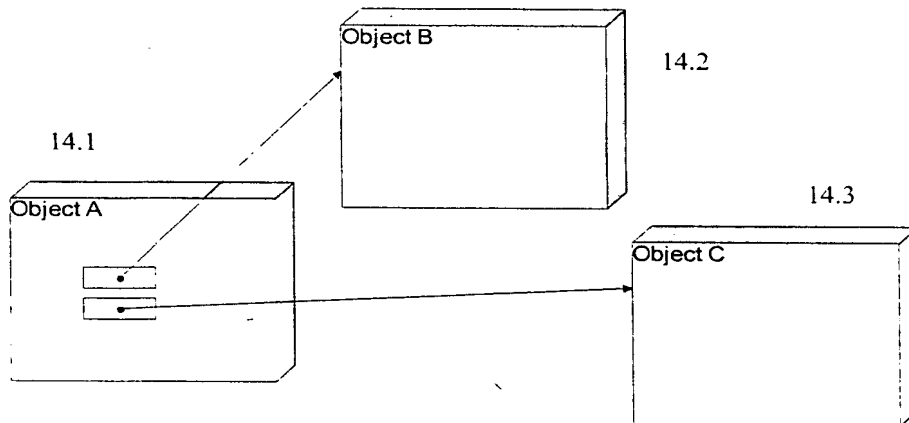


FIGURE 14

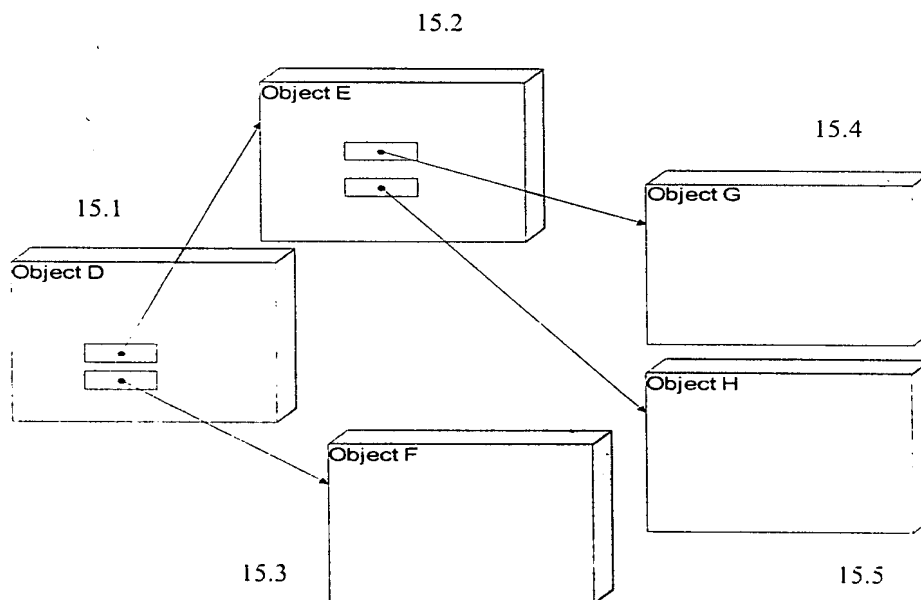


FIGURE 15

(Signature)

Anand and Anand, Advocates,
Attorney for the Applicants

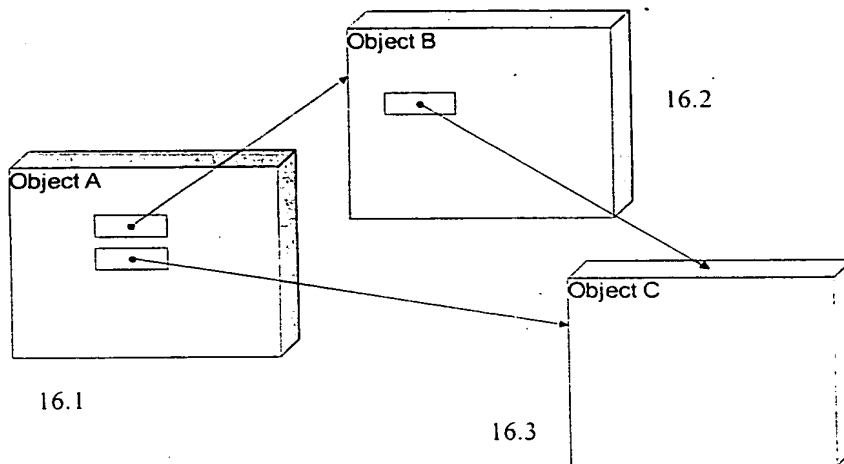


FIGURE 16

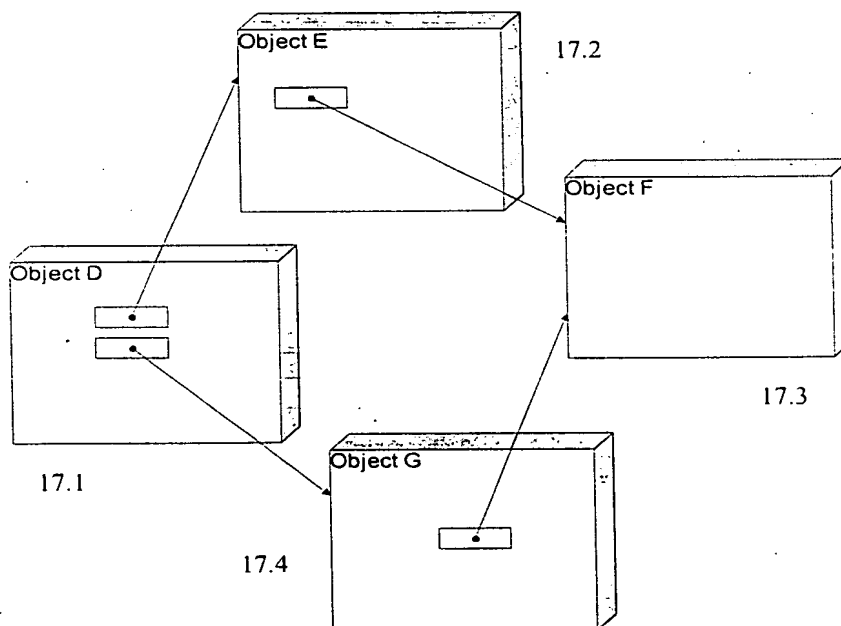


FIGURE 17

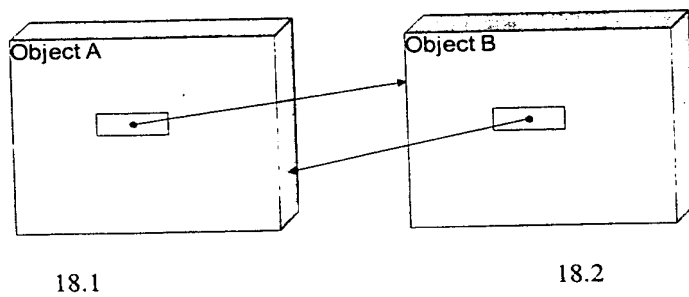


FIGURE 18

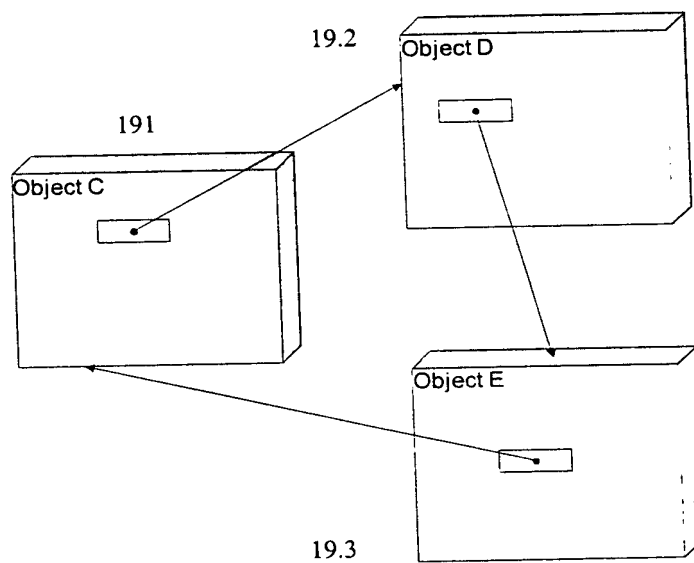


FIGURE 19

Anand

Anand and Anand, Advocates,
Attorney for the Applicants

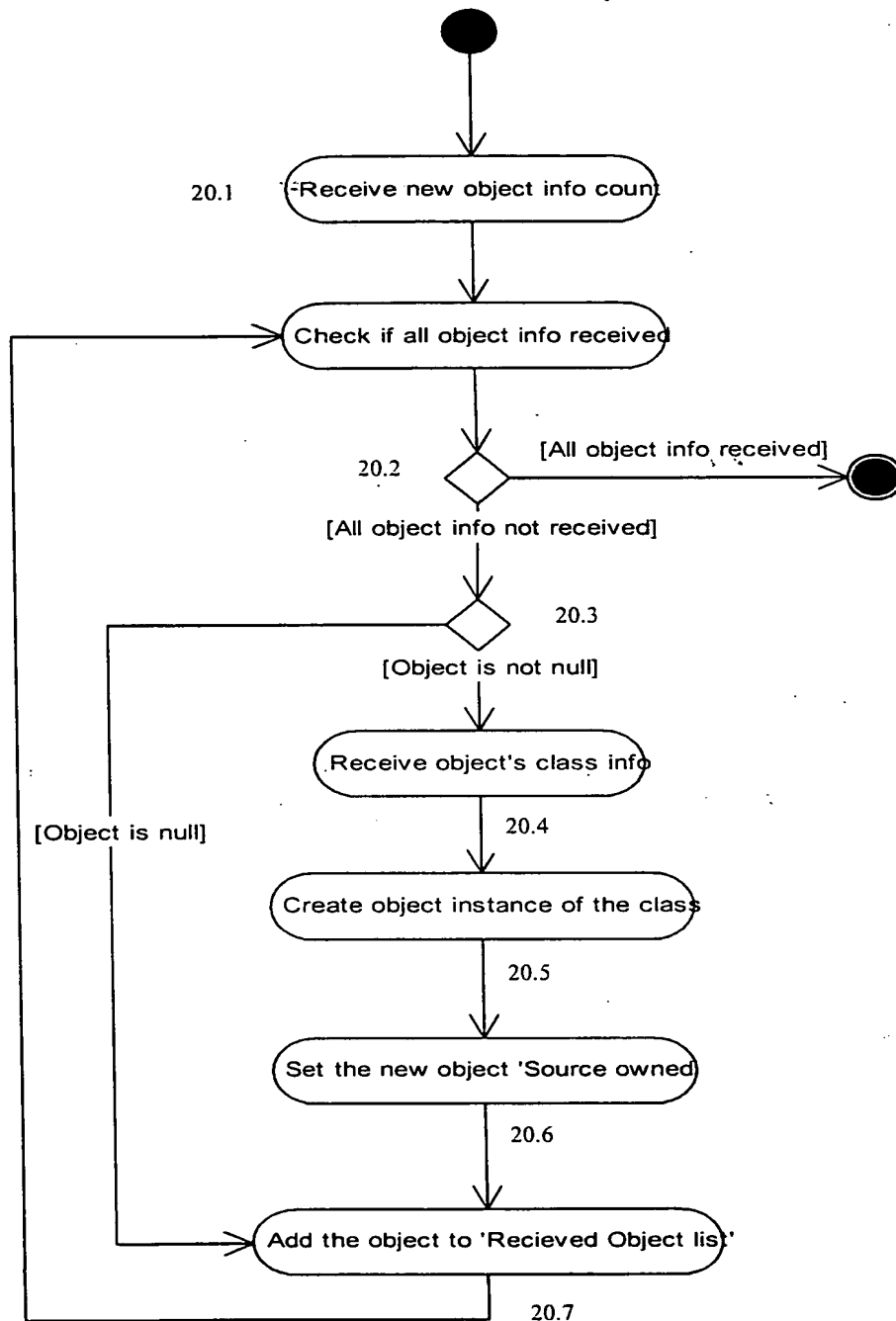


FIGURE 20

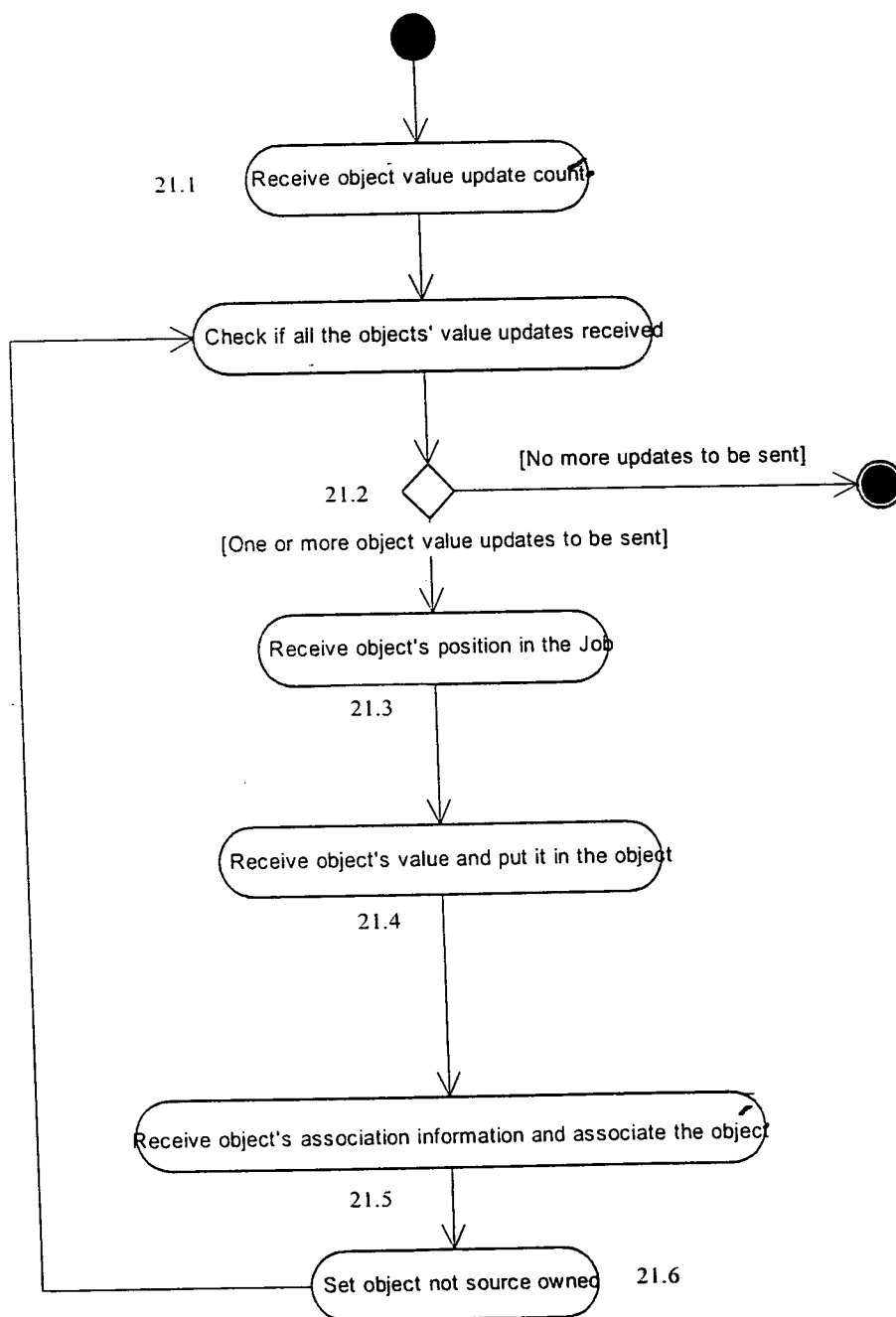


FIGURE 21

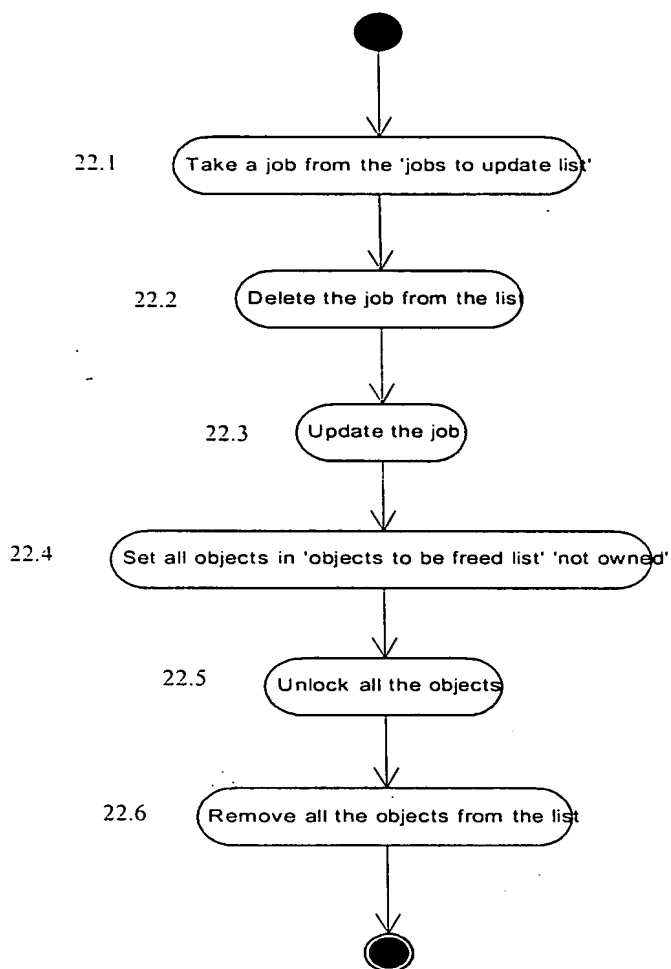


FIGURE 22

Anand
Anand and Anand, Advocates,
Attorney for the Applicants

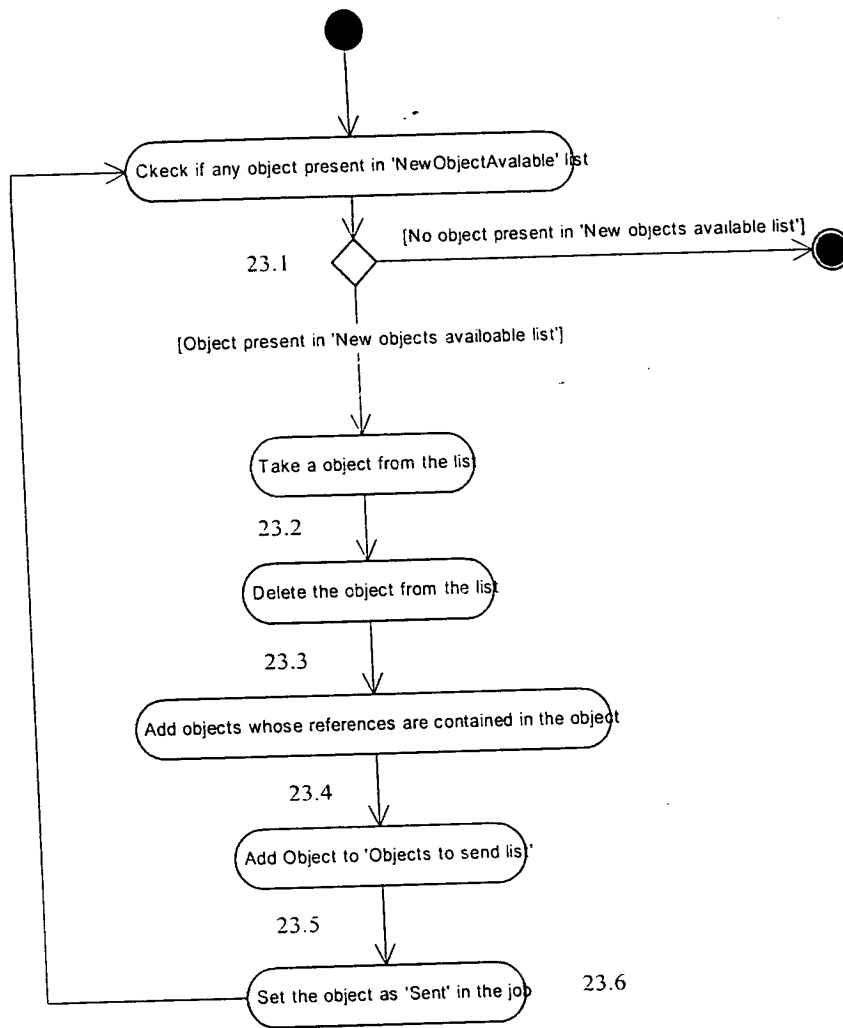
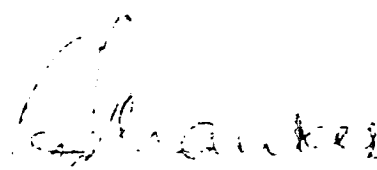


FIGURE 23


Anand and Anand, Advocates,
Attorney for the Applicants

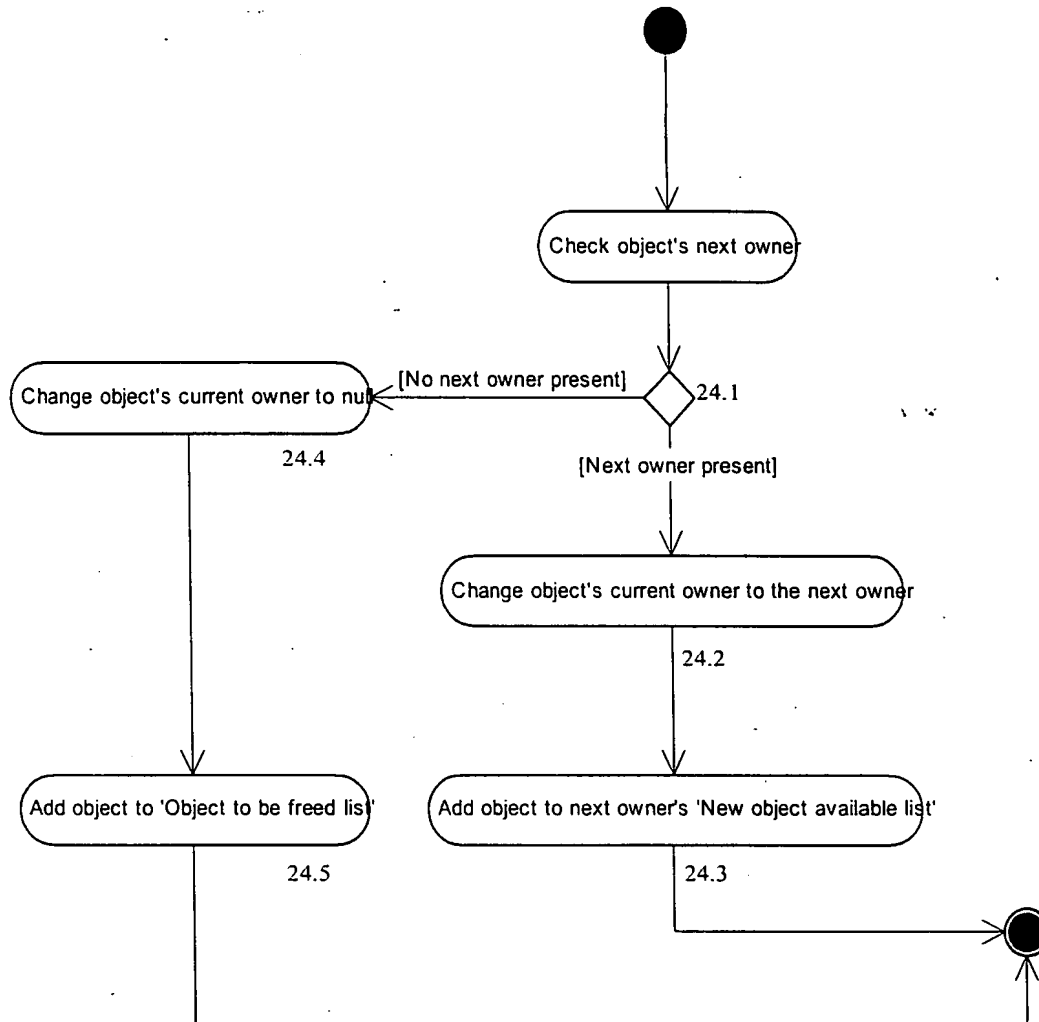


FIGURE 24

Anand
Anand and Anand, Advocates,
Attorney for the Applicants

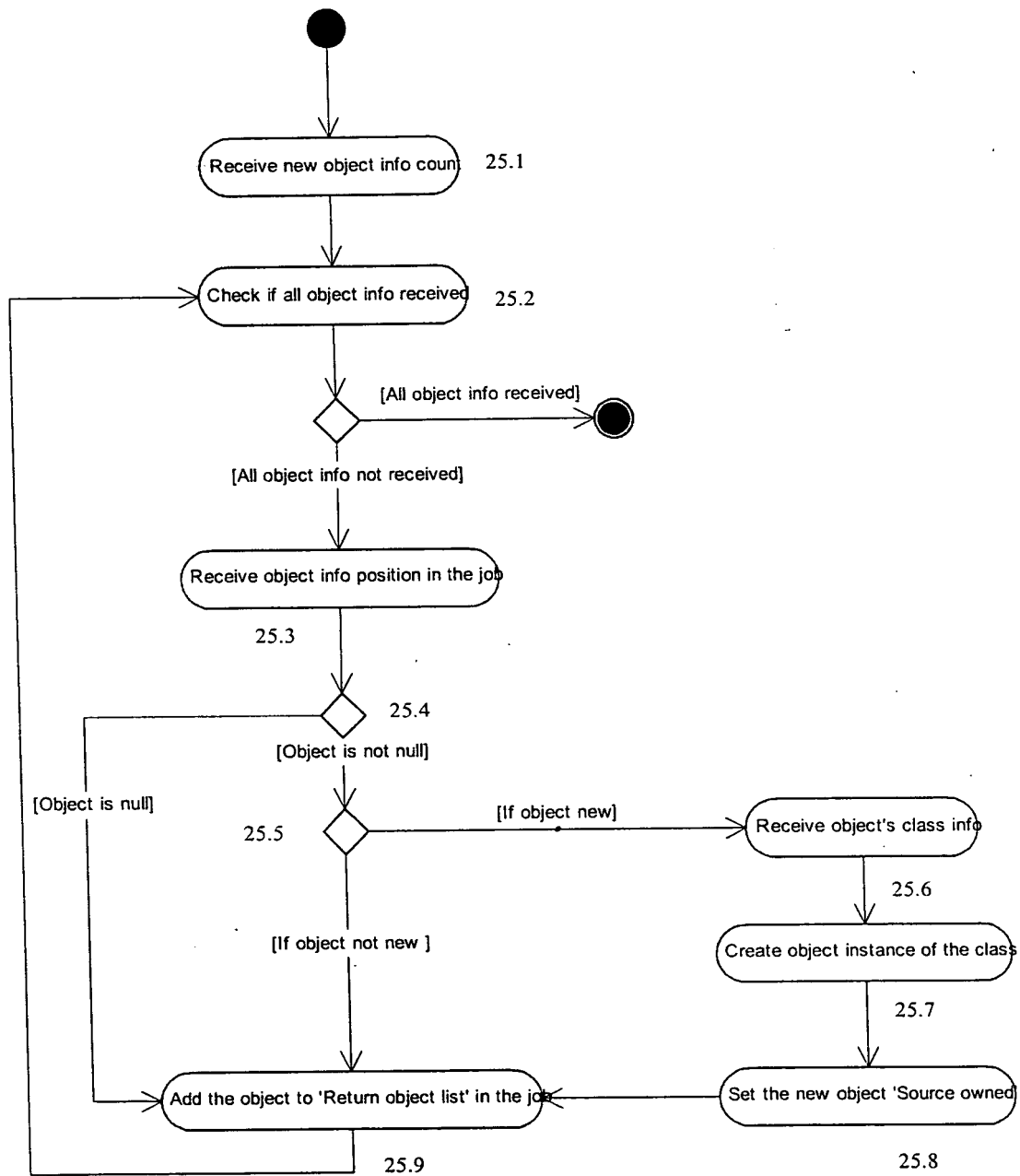


FIGURE 25

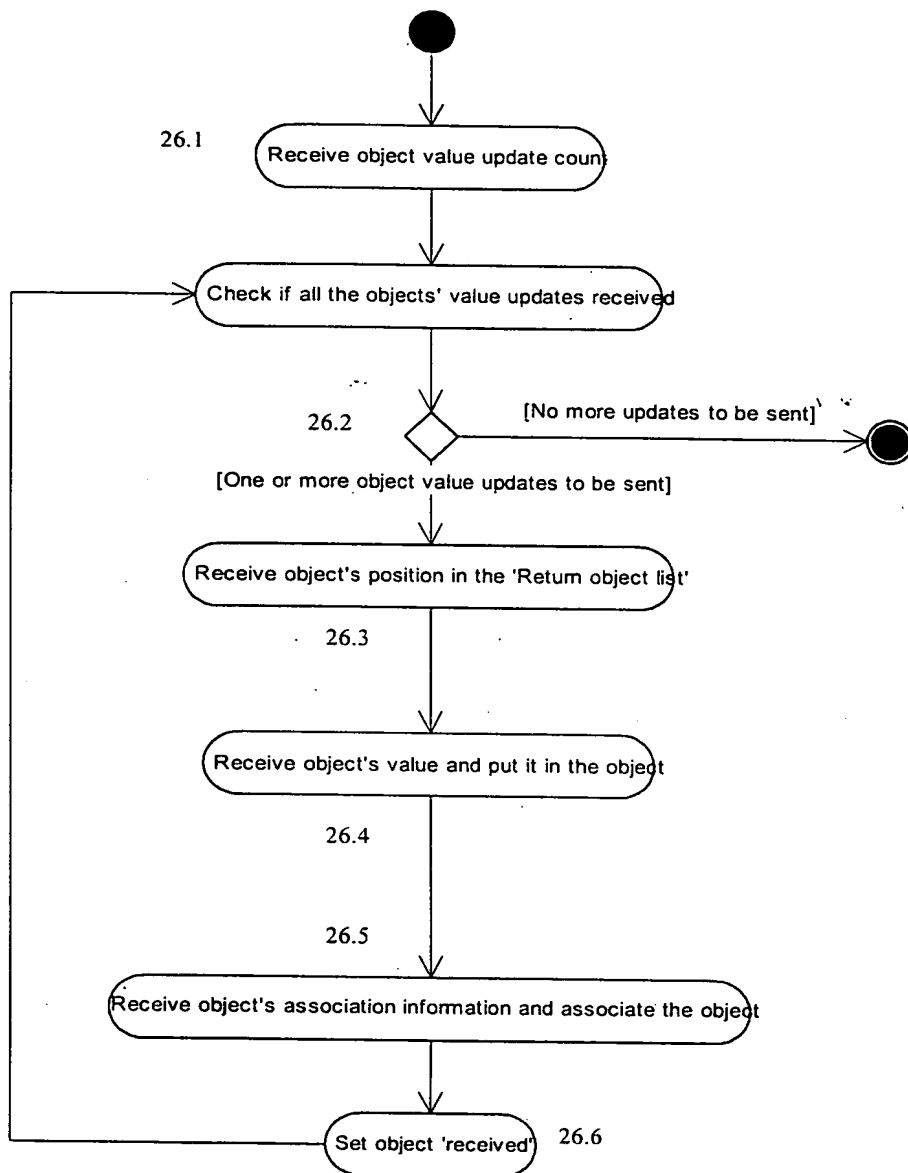


FIGURE 26

Anand and Anand

Anand and Anand, Advocates,
Attorney for the Applicants

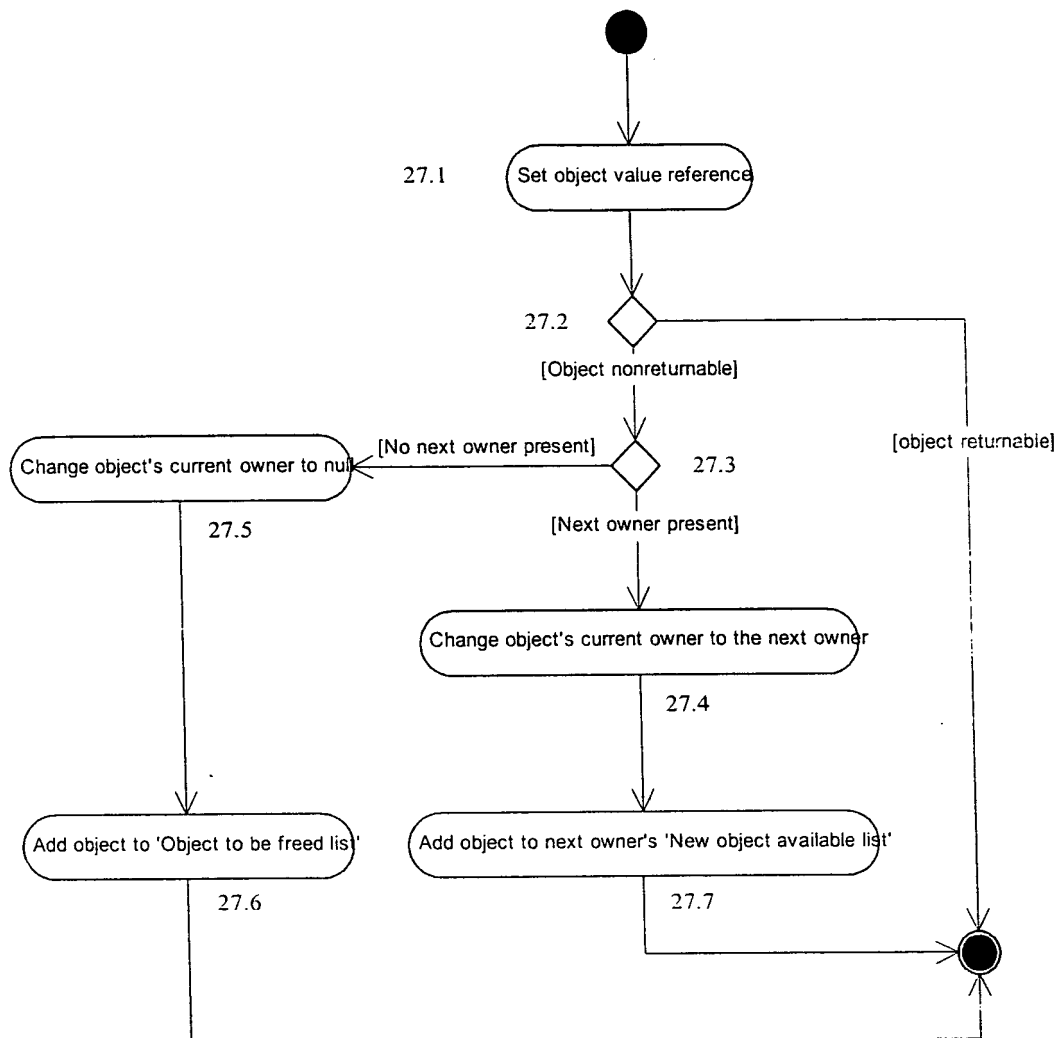


FIGURE 27

Anand and Anand, Advocates,
Attorney for the Applicants

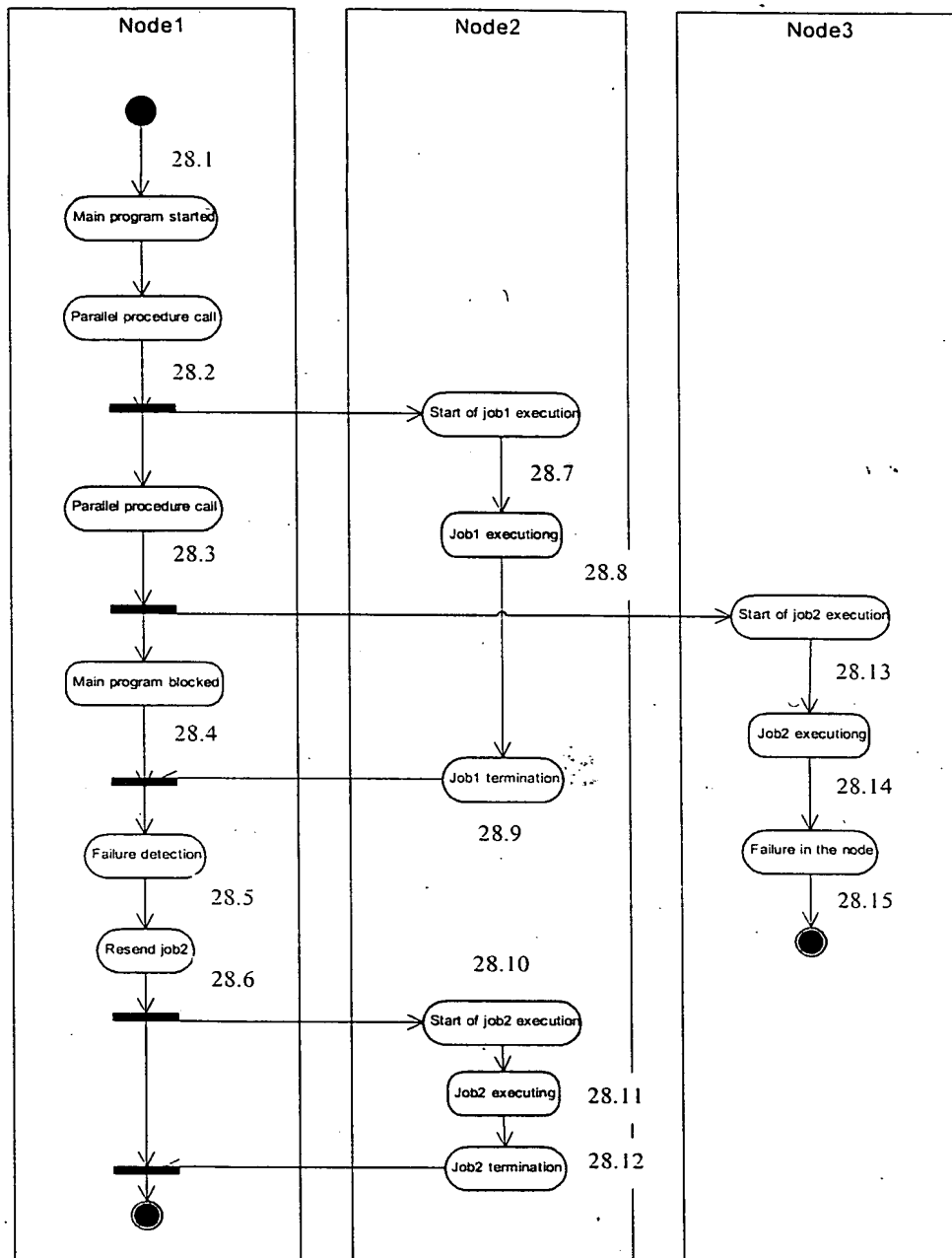


FIGURE 28

Anand and Anand
Anand and Anand, Advocates,
Attorney for the Applicants

0834-2

29 AUG 2002

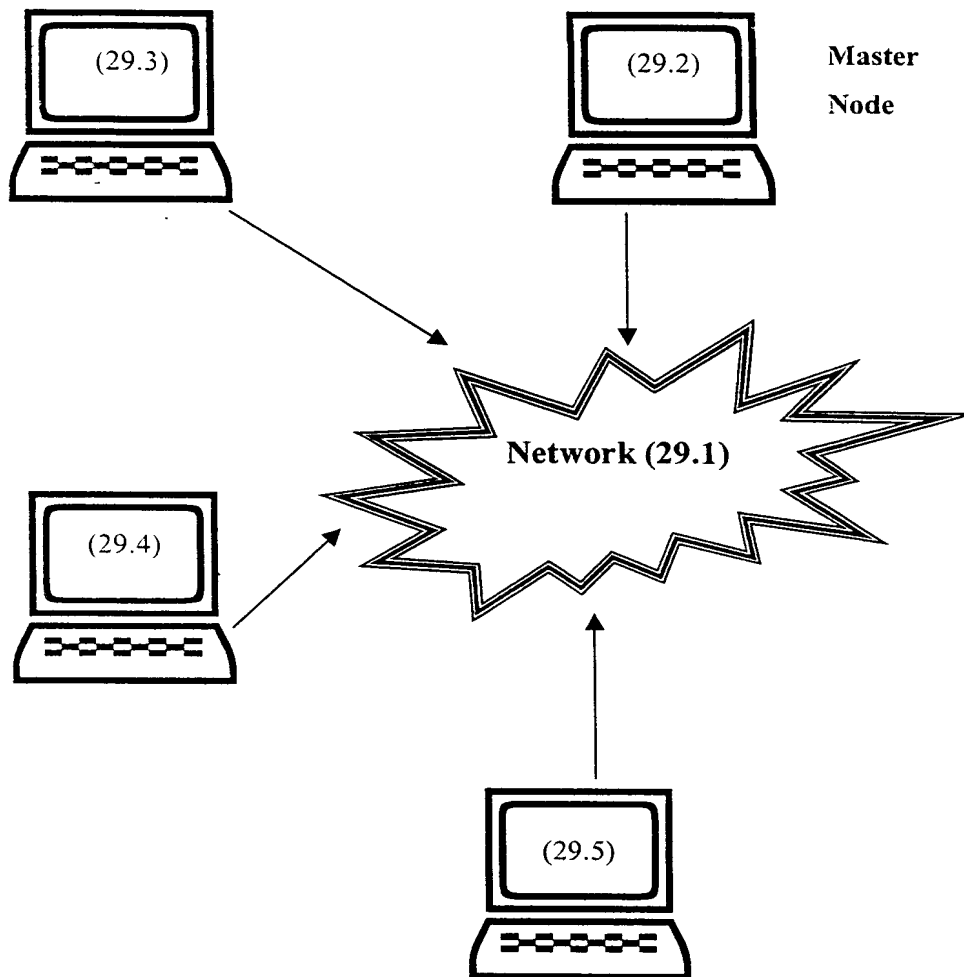


FIGURE 29

DUPLICATE

Anand and Anand, Advocates,
Attorney for the Applicants

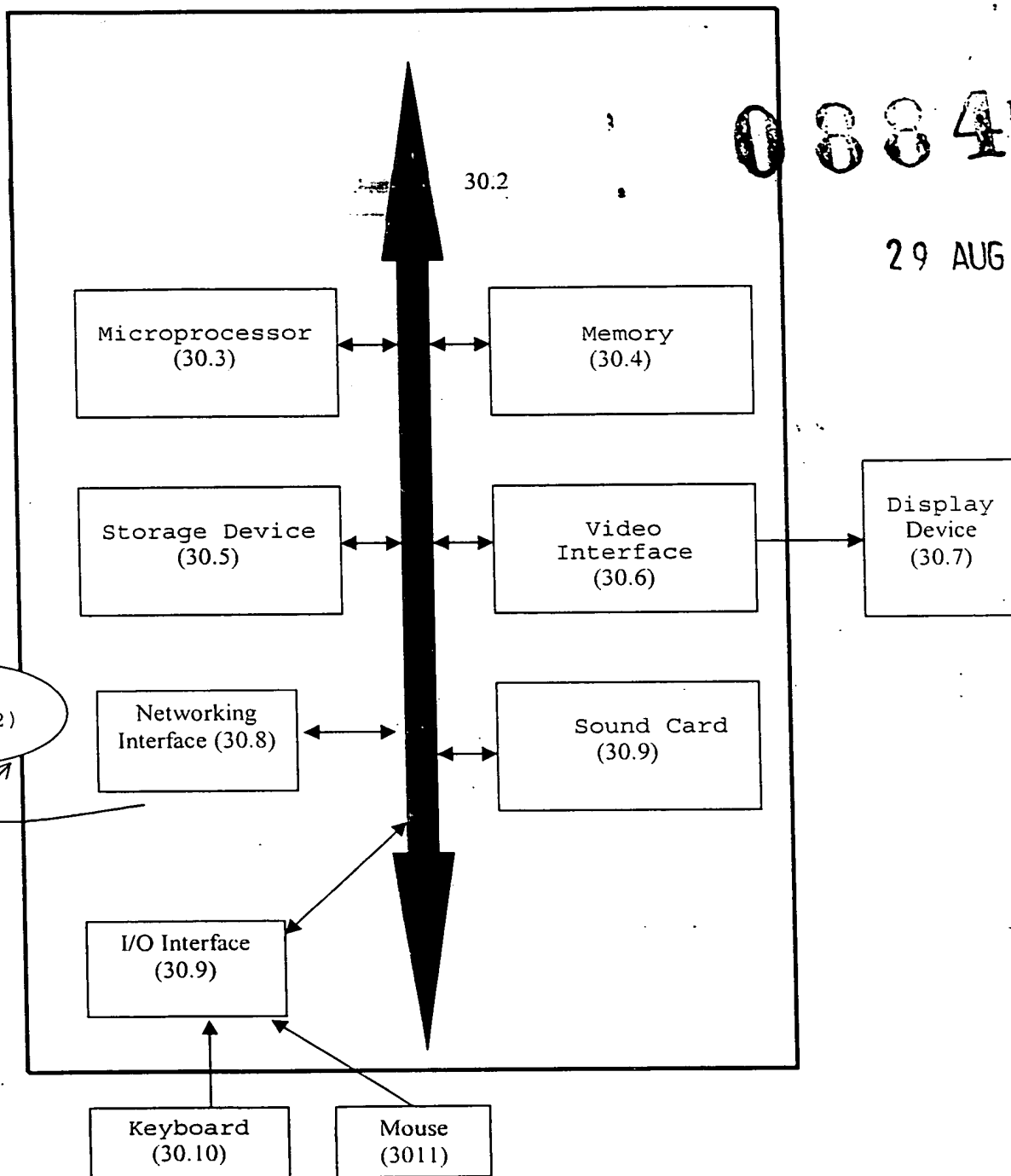


FIGURE. 30

Anand and Anand
Anand and Anand, Advocates,
Attorney for the Applicants

THIS PAGE BLANK (USPTO)

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

☐ **BLACK BORDERS**

☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**

☐ **FADED TEXT OR DRAWING**

☒ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**

☐ **SKEWED/SLANTED IMAGES**

☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**

☐ **GRAY SCALE DOCUMENTS**

☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**

☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**

☐ **OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.

THIS PAGE BLANK (USPTO)